Universitetet i Oslo
Institutt for informatikk

An Evaluation of the ABEL System

Martin Thornquist

Cand Scient Thesis

25th January 2002

An Evaluation of the ABEL System

Martin Thornquist

25th January 2002

Preface

This is a thesis to the cand.scient. degree, submitted to the Department of Informatics, University of Oslo. It is a part of the ABEL project at the department.

I chose to join the ABEL project after three years of studies at the University of Oslo, centered around computer science/programming, and after discovering an interest for logic and the formal side of computer science. At the time it seemed quite a challenge, but also the most interesting work I could find. My work proved to be both of these, but I hope I mastered it sufficiently in the end.

I have chosen to write in English, even though I am a native Norwegian, and the previous theses in the project has been written in Norwegian. This has several reasons; first of all, I believed myself to be capable of mastering it, and wanted to see how it worked out. Secondly, most literature other than the mentioned theses are in English.

I want to thank my advisors. Olaf Owe, always forthcoming, always willing to explain at length. And Ole-Johan Dahl, one of the pioneers of computer science in Norway. Although retired before I finished, it has been a priviledge to have him as advisor. Dahl and Owe were also responsible for the wakening of my interest in the field of verification, through their teaching of the course IN 217 (Program Specification and Verification) at the Department of Informatics.

I also want to thank Tore Jahn Bastiansen for answering all my questions in the beginning and helping me get started using the ABEL system, and Steinar Midtskogen and Olav Andree Brevik for explaining various things about the system during my work.

Last, but not least, I want to thank my family for supporting me all the way, even though it did take a bit more time than I hoped at the start.

Oslo, 25th January 2002
Martin Thornquist

Contents

Chapter 1

Introduction

ABEL is short for *Abstract Building, Experimental Language*. That is, the system includes a theorem prover, but is more than that; it is first and foremost meant to be a programming environment. Secondly, *Experimental* signifies that ABEL does not primarily aspire to become a mature tool for practical programming, but rather be in continuous development as a vessel for research into the areas the project involves. Most recently the research has been based around constructing an interactive first order logic theorem prover, for reasoning about programs. The reasoning is based on Ole-Johan Dahl's work as detailed in [Dah92], with a few minor differences—most of them due to the fact that Dahl's work is intended for human, not computer-based, reasoning.

1.1 Focus and Goal

My focus in this thesis will be on evaluating the current ABEL system, to provide background to better be able to decide how to improve the system: whether there are outright errors, if some things should be changed to help efficient proof construction, and how to better support automation of proof construction in the system. I will look at the system as a verification system, and in that context my goal is to point out what directions further work on the system would be most reasonable to pursue, or alternatively what could be done better if designing a new system.

1.1.1 Why Automate?

In several parts of this thesis, my focus will at least partly be on how better to support automating proof construction and, ultimately, program verification. The reason for this focus is, of course, that the overall goal of computer-assisted proof construction for program verification is greater efficiency, and there is the most to gain in that respect in making the com-

puter do more of the proof work. I find the ABEL system to function quite well (except for a few details of the user interface) for proof checking, so the greater potential for improvement must lie in the proof construction.

1.1.2 What this Thesis does not Discuss

I will not focus on minor problems and shortcomings with the user interface. I will to some degree describe fully automatic proof systems, but not focus on issues that are of little or no consequence to interactive systems. As we in a system for verification want to automate as much as reasonably possible of the proof process, automatic systems are interesting—but issues pertaining to problems that are best left to the user, are not. However, I will discuss whether it is reasonable to try to subject certain areas of proof construction to automation.

1.1.3 On the Contents of this Thesis

I have partitioned this thesis into six main parts: chapter 2 reviews some of the logical and mathematical theory that is necessary to understand the workings of theorem provers, and a few different techniques used in mechanised theorem proving. Chapter 3 is a user's guide to the ABEL proof system—something I missed when I first started my work on the project. Chapter 4 contains a motivational test run of a number of theorems with the prover, where I try to get an impression of the strengths and weaknesses of the ABEL system. Chapter 5 contains some further, larger examples of theorem proving with the ABEL system, as a background for the heart of this thesis: chapter 6, where I evaluate the current system, and discuss its usability as it currently stands. Finally, in chapter 7, I summarise and conclude on the presented matter. Additionally, I have included as appendices two standard modules from the ABEL system that are used in the included proofs, and a couple of further examples of proofs generated by the system, one of which is explained in regard to how to use the prover.

Chapters 2 and 3 should give sufficient background for understanding the rest of the thesis for a reader unfamiliar with theorem proving in general and/or the ABEL system specifically.

1.2 The ABEL Language

The ABEL language is based on the (as yet unimplemented) programming language developed by Ole-Johan Dahl for his research and teaching, and also used in some of his publications, including [Dah92]. It is a language with all the basic structures of traditional imperative languages (with a syntax resembling that of the Algol family), but it also includes a part for spe-

cification, and an extensive applicative type generation system more akin to those of functional languages like ML and algebraically based ones.

The most up-to-date reference of the language is probably to be found in [Bas95], p. 87. Brevik includes a reference for the expression language in [Bre98], p. 36.

1.2.1 The ABEL System

The current incarnation of the ABEL system consists of the base system, including parser and type checker, written by Tore Jahn Bastiansen [Bas95]; a proof system, written by Olav Andree Brevik [Bre98]; and a rewriter, written by Leif John Korshavn [Kor98] and Steinar Midtskogen [Mid99]. Furthermore, Bastiansen has worked on the system as part of his as yet unfinished doctorate thesis. Some articles on the ABEL system, and related topics, may be found from the ABEL project's home page, [abe].

1.3 Logical Reasoning

As an interactive proof system, the ABEL system expects the user to understand the proofs the system generates; hence, the system must perform proofs in a way similar to how humans do. Thus, our tools for proving theorems are those of formal logical reasoning: deduction, induction and term rewriting.

Our tool for handling logical constructs, such as $\wedge$ (logical and), $\vee$ (logical or), and $\Rightarrow$ (logical implication), is deduction. We use a system of deduction rules; a set of rules for transforming each of the logical constructs. The ABEL system uses the backwards proof construction (BPC) system, which is described below.

The method of induction should be known to the reader; as ABEL is a typed language, we use this tool for handling typed variables and similar through generator induction.

Finally, term rewriting is a technique for transforming one expression into another, hopefully simpler. Rewriting uses a set of rules for how to transform expressions, and searches in this set with matching algorithms to find a matching rule for a certain expression. This set of rules is dynamic; rules are added as functions and lemmas are read and parsed by the system.

1.4 Terminology

Some of the terminology used in this thesis may be unknown to the reader, so I will try to explain the most significant terms in the following.

1.4.1 Sequent

A *sequent* is the basic unit of a proof, consisting of the *antecedent* (the assumptions) and *consequent* (the theorems). We depict it thus:

$$\text{sequent: antecedent} \vdash \text{consequent}$$

We call $\vdash$ the sequent sign. The antecedent and consequent are (possibly empty) sets of clauses.

As an example, assume that at some point in a proof we have the following situation: assuming a set of clauses F we conclude the set of clauses G. The clauses are separated by semicolons. This is depicted as a sequent like this:

$$F \vdash G$$

1.4.2 Proof Obligation

When verifying programs with Hoare logic we use a set of rules to generate a set of expressions to be proved, called proof obligations, for each language construct (many of the rules for the ABEL language are described in [Dah92], and an exhaustive list is given in [Lin99] pp. 60-66).

1.4.3 BPC

BPC is short for Backwards Proof Construction, which is the method of choice for logic reasoning both in [Dah92] and the ABEL system. The other method for reasoning described in [Dah92] is natural deduction (ND), or forward proof construction. BPC starts with the theorem at hand and applies deductive rules backwards until all proof branches are trivially satisfied by axioms of the deductive system, while ND starts with the axioms and tries to construct a proof ending with the theorem that is to be proved. I will expand on this in chapter 2.

1.4.4 Generator and Observer Functions

We separate the functions in a type module into two groups, generators and observers. Generators are functions that generate (and return) a new value of the type. E.g. for integers, 0, S (the successor function) and + are all generators for the integer type. Observers, on the other hand, take as argument a value and return a decision for some question, (usually) as a Boolean value. A function which decide if an integer is positive is an observer for the integer type.

1.4.5 Proof Construction vs. Proof Checking

These are central terms in the matter at hand. Proof *construction* is, naturally, the process of constructing a proof, by choosing which rules should be applied at each point, etc. This is what an automatic theorem prover does. Proof *checking*, on the other hand, is the process of checking if a completed proof is correct. Interactive theorem provers perform a combination of proof construction and proof checking. Checking a proof is algorithmically easy and can be done in polynomial time, while constructing a proof is very hard—actually, as we will see in chapter 2, proof construction is algorithmically impossible; it is not possible to construct an algorithm that will construct a proof for every true theorem.

1.4.6 ABEL vs. the ABEL System

ABEL is the name of the language, not the concrete system. I will therefore use ABEL to mean the language, and *the ABEL system* to mean the present implementation of the verification system.

1.4.7 Theorem vs. Expression

I will use both the terms *theorem* and *expression* for the logical statements a theorem prover is employed to prove. I will try to stay true to the normal mathematical convention regarding the use of *theorem*, in that I will use *theorem* to mean the statements we invoke a prover to prove (and which we believe to be true), and *expression* to mean statements encountered during a proof, sometimes for which the matter of truth is more doubtful.

1.4.8 Sequence Operators

Not strictly terminology, but as sequences are used in many of the examples I will describe the most important of the sequence operators we use.

ϵ is the empty sequence.

$\vdash$ is right concatenation, the basic generator of sequences in ABEL. Sequences in ABEL are visualised as growing towards the right, as opposed to the more often used left growth of e.g. the functional languages ML and Haskell.

$\dashv$ is left concatenation; as right concatenation is the sequence generator, $\dashv$ adds to the end of a sequence.

$\vdash\!\!\dashv$ concatenates two sequences.

$\#$ is the length of a sequence.

1.4.9 ASCII Representation

As we use ASCII text for communicating with the ABEL system I have included the ASCII representation of the different operators in the following table:

Symbol	ASCII	Symbol	ASCII
$\vdash$	`|-`	$\dashv$	`-|`
$\vdash\!\!\vdash$	`|-|`	$\dashv\!\!\vdash$	`||--`
ϵ	`e`	#	#
$\forall$	`forall`	$\exists$	`exist`

Chapter 2

A Review of Theory

There is no known algorithm for constructing a proof for a theorem; if there had been, much of the research into theorem proving would be moot, as it would just be a question of efficiency. Rather, it is known that the problem of deciding whether an expression is a theorem—often referred to as *THEOREMHOOD*—is undecidable: it is actually provably not possible to design an algorithm that is able to decide in general whether an expression is a theorem of an axiomatic system of sufficient complexity.

2.1 Undecidability of Proving Theorems

Now, there may seem to be a way out of these problems in the last sentence of the above paragraph: "an axiomatic system of sufficient complexity". Does this then imply that we can use a system of less complexity? It has been shown that any system of "interesting" complexity, among them prominently number theory, has this flaw.

The theory of undecidability stems from the work of Kurt Gödel, arguably one of the most brilliant mathematicians of the 20th century. In 1931 Gödel published his results, showing that in any system of the aforementioned sufficient complexity there are truths that cannot be proved inside the system. (See [Gö31] for the original article, or e.g. [NN58] for a popularised explanation.) This is the reason why automatic theorem provers have proved so hard to make: one cannot rely on logical reasoning alone to complete the proofs, and hence the completion of most automatic systems has hinged on the development of useful artificial intelligence—and indeed been one of the most often used example problems in artificial intelligence research. The failure of artificial intelligence research to produce usable AI has in turn led to a shift of focus towards interactive provers, geared mostly towards helping the user with the tedious parts and the bookkeeping of proving (and guaranteeing that no errors are made), instead of trying to do the whole proof autonomously.

There are several pitfalls a theorem prover must try to avoid. The perhaps simplest to deal with of the results of the undecidability of THEOREMHOOD is that a proof might stop with the computer unable to find a way to continue the proof. A rather more sinister problem is that of infinite recursion—i.e. the problem that there is no known method of discovering aforehand if a line of reasoning will not turn into a useful result, but rather continue forever.

Thus, it seems that to be able to construct proofs—or parts of proofs—autonomously, a computer program must have the ability to backtrack when a proof branch does not yield any useful result, and to somehow avoid or stop infinite recursion. An obvious example of the first from the ABEL system is that the bpc strategy can entail use of the tall rule, which removes a universal quantifier from the theorem part of the sequent. If a line of reasoning does not work out after having used this command, one might instead want to try induction over the variable with the now removed universal quantifier. To avoid needing user intervention, the system then has to be able to backtrack to the position before the use of tall, or reintroduce the universal quantifier. An example of the latter is that the bpc strategy only incorporates those of the BPC rules that are constructive; that is, they lead to a syntactically simpler expression, so any recursion will stop when the expression cannot be simplified any more. I will discuss further these and other problems later in this chapter.

2.1.1 Are Programming Languages Special?

One might surmise that programming languages can be made that are simple enough to not be inherently incomplete. However, what we usually mean by programming languages today is Turing-complete languages, which means they possess enough complexity to fall under the class of incomplete systems. Specifically, all modern programming languages have facilities for unbounded recursion, and as shown in e.g. [Hof79], a system incorporating unbounded recursion must be incomplete.

2.2 Proof Techniques

Several proof techniques has been developed and employed for mechanised proof construction. I will here give a short overview of the primary techniques.

2.2.1 Resolution

Resolution is commonly considered the most effective method for automatic theorem proving, but does not resemble the way humans do proofs, and thus is not very well suited for interactive systems.

$$T\wedge \;:\; \frac{\vdash P;\ \vdash Q}{\vdash P\wedge Q} \qquad A\wedge \;:\; \frac{P,Q \vdash R}{P\wedge Q \vdash R}$$

$$T\vee \;:\; \frac{\vdash P,Q}{\vdash P\vee Q} \qquad A\vee \;:\; \frac{P \vdash R;Q \vdash R}{P\vee Q \vdash R}$$

$$T\Rightarrow : \frac{P \vdash Q}{\vdash P\Rightarrow Q} \qquad A\Rightarrow : \frac{\overline{P} \vdash R;Q \vdash R}{P\Rightarrow Q \vdash R}$$

$$T\neg \;:\; \frac{P \vdash}{\vdash \neg P} \qquad A\neg \;:\; \frac{\vdash P}{\neg P \vdash}$$

$$T\forall \;:\; \frac{\vdash P^x_{x'}}{\vdash \forall x\cdot P} x' \text{fresh} \qquad A\forall \;:\; \frac{P^x_t,\forall x\cdot P \vdash Q}{\forall x\cdot P \vdash Q}$$

$$T\exists \;:\; \frac{\vdash P^x_t,\exists x\cdot P}{\vdash \exists x\cdot P} \qquad A\exists \;:\; \frac{P^{\prime x}_x \vdash Q}{\exists x\cdot P \vdash Q}\ x' \text{ fresh}$$

Figure 2.1: The BPC rules, as implemented in the ABEL prover.

2.2.2 Model Checking

Model checking consists of constructing a model of the problem, and then testing this model—for (untyped) predicate logic, model checking is equivalent to proof by truth tables.

Model checking is effective when concerned with only a small number of finite states, but in general the search space increases exponentially with the complexity of the expression. However, combined with other proof techniques (which reduce the complexity before model checking is applied), model checking has shown promise. Brevik has performed some testing with the ABEL system, described in [Bre98] p. 111. He did not have the time to implement a general model checker, but he tested out something resembling truth tables for predicate logic, using the *BCUT* inference rule:

$$BCUT : \frac{\vdash P^x_{\mathbf{true}};\ \vdash P^x_{\mathbf{false}}}{\vdash P}$$

Brevik tested this rule with the proof for associativity of equivalence for three to 20 variables, and compared this to using BPC rules. He found model checking to be significantly faster, and a very promising technique for interactive proof systems based on inference rules.

Model checking has also been implemented successfully in PVS, see [Sha96a] and [Sha96b].

2.2.3 BPC

Backwards proof construction (BPC) is, as mentioned in the introduction, the method of choice for logical deduction in the ABEL system's prover. The complementary method, called natural deduction (ND), is a system

for formal reasoning where one starts from the axioms of the system and tries to build a proof ending with the theorem to be proved. ND is not equivalence preserving, while BPC, in contrast, is.

It should be quite obvious that BPC lends itself rather more to mechanical proof construction than does ND: programming a computer to find the right starting axioms for ND and making the right choices along the proof path to end up at the desired theorem seems to be a very, very hard task, as opposed to starting with the theorem to be proved and trying to simplify it. Another, more concrete, obstacle to the use of ND is that the BPC rules (except for two special cases) simplify the sequent, and thus will not recurse endlessly—while ND constructs an ever more complex sequent if not hitting the theorem to be proved. Thus, a system based on ND would need some kind of heuristic to stop a proof branch that seemingly would not lead to the theorem being proved. However, such a heuristic would necessarily never be perfect—one can generally never be sure that a proof branch will not be successful, as the rules that are non-constructive in BPC is simplifying in ND. The BPC rules are listed in figure 2.1; the ABEL system's `bpc` strategy consists of all those except $T\exists$ and $A\forall$.

Non-constructive BPC rules

The rules $T\exists$ and $A\forall$ in figure 2.2.3 are called non-constructive: applied as BPC rules they make the sequent more complex, rather than simplify it, and may therefore lead to infinite recursion in a proof. This should be obvious from the definitions, as the rules keep all the parts of the previous sequent, as well as adding another. Because of this, the `bpc` strategy cannot incorporate these rules; indeed, a strategy incorporating those rules would have to be very carefully constructed regarding to how it applies them, making sure not to recurse infinitely.

2.2.4 Induction

The induction principle is central in ABEL. All data types are defined by inductive generator functions, and generator and observer functions (i.e. functions operating on variables) are usually defined by generator induction with the `case` construct.

Generator Induction

Although considered one of the most advanced among proof techniques, induction is relatively easy to implement. The following equation shows the general generator induction principle:

$$\frac{\begin{array}{c}\vdash \forall(c : C) \circ F(c) \ ; \\ F(x) \vdash \forall(g : G) \circ F(g(\ldots, x, \ldots))\end{array}}{\vdash \forall(x : T) \circ F(x)}$$

Here C is the set of constant generators for the type T, and G is the non-constant ones. The example that most easily comes to mind is the natural numbers; here the only constant generator is 0 and the only non-constant S, the successor function. Induction over the natural numbers can be represented as in the following equation:

$$\frac{\begin{array}{c}\vdash F(0) \ ; \\ F(x) \vdash F(S(x))\end{array}}{\vdash \forall(x : Nat) \circ F(x)}$$

As shown, simple generator induction consists of proving the expression first with all constant generators (that is, generators that do not take arguments, such as 0 of the natural numbers) substituted for the variable under induction, and then, assuming the expression being proved, proving it with all non-constant generators applied to the variable under induction.

As we will see in chapter 4, induction and rewriting suffice to prove many useful theorems.

Generalising a Variable

Even if a variable has lost its universal quantifier, e.g. due to application of the $T\forall$ BPC rule, it still might be possible to use it for induction. This may be accomplished by modifying the induction module to be able to do generalisation automatically, or by providing a separate generalisation facility which (re)introduces a universal quantifier after testing that the variable is indeed eligible.

One problem to be aware of if implementing a separate generalisation facility, is that of divergence: if induction is applied repeatedly to the same variable the sequent will grow endlessly. In other words, generalisation is not constructive, and should be treated with similar caution to the BPC rules $T\exists$ and $A\forall$. However, this is not very difficult to prevent, if one has a mechanism to make sure to apply induction to a certain variable no more than once.

2.2.5 Rewriting

Rewriting is a technique many theorem provers employ, whether otherwise built around a single method of proof construction, like the earlier automatic systems, or several methods, like PVS and the ABEL system.

Rewriting simplifies an expression by substituting subexpressions for (hopefully) simpler, equal expressions. Rewriting of expressions is based on a convergent system of rewrite rules. To be convergent, the system has to be terminating; it should not be possible to construct an infinite sequence of rewrite rules. Furthermore, the system has to be confluent, that is, if several different rules are at some point applicable, the same expression (called normal form) should be obtained regardless of the order in which the rules are applied. More information on term rewriting may be found in e.g. [Kir].

Some interactive systems, e.g. PVS, perform rewriting automatically after changes in the sequent has occurred. However, this would lead to problems if implemented naìvely in the ABEL system, as I have in my tests encountered proof branches where rewriting destroyed the only possible route to success.

2.2.6 Strategies/Tacticals

As interactive systems were developed, it became apparent that a method for combining the basic commands would be valuable. As we will see in chapter 4, a quite simple command combination can complete proofs for many simpler theorems. A completely general command combination would of course amount to an automatic system, which we have seen has proved very hard to make, but substantial parts of proof can often be completed with merely one relatively simple command combination.

These combination methods are usually called tacticals or strategies, and are used in all major current interactive provers. In the ABEL system we call them strategies, and this mechanism has constructs to facilitate elaborate proof schemes with operators to serialize commands, repeat, and apply rules at multiple positions in a sequent. Brevik has discussed the syntax of strategies in greater depth in his thesis [Bre98], but for easy reference the following BNF schema excerpt details the syntax we use for strategies in the ABEL system.

<cmdline>	::= <cmdlist> | (<cmdlist>) | [<cmdlist>]
<cmdlist>	::= <cmd> [, <cmdlist>]*
<cmd>	::= RULENAME <mposspec> <eqspec>$^?$
	| repeat <cmdline>$^?$ POS$^?$
<mposspec>	::= <posspec> multi$^?$
<posspec>	::= [-$^?$ POS]$^+$ | + | - | *
<eqspec>	::= '<hlexpr>' | #-$^?$ POS | $

RULENAME is some specific rule or command, POS is a position specification as described below. <hlexpr> is an expression in the host language.

Commands listed in parentheses are applied in order; those in brackets are applied only until one is successful. Positions (where in the sequent the rule should be applied) may be given as specific numbers: -1 signifies the first antecedent, +2 (+ is optional) the second consequent; or as +, signifying any consequent; - signifying any antecedent; and * for any part. multi makes a rule be applied in all possible parts of the sequent (without multi, a rule is allied only to one part, regardless if it is applicable to more), while repeat repeats a rule as long as it is successful.

An example of a very simple strategy—in fact, as we will see, one that is able prove many of the theorems in chapter 4—is

```
simpleproof ==
  strategy
    (induct 1, rewrite, xrewrite)
  endstrategy
```

Another, only slightly more advanced and general example for inductive proofs (which would also have proved all the theorems the above did) is

```
inductiveproof ==
  strategy
    repeat [xrewrite, induct +]
  endstrategy
```

2.3 Other Theorem Provers

There has been made a number of theorem provers, not all of them specifically for verification, and with varying degrees of success. I will in the following mention some of the most influential and interesting, and describe a few in somewhat more depth.

Most of the systems were made purely for research, often employing metalanguages (the languages the theorems, lemmas etc. are written in) that were designed to be simple to implement, not necessarily to use, and many of them would indeed be cumbersome or awkward in practical use. However, this is not always the case; it is to be noted that ML, one of the most prominent of the functional programming languages and the language the current incarnation of the ABEL system is implemented in, originated as metalanguage for a theorem prover written by Larry C. Paulson (indeed, ML is an acronym for Meta Language). This prover is a predecessor of Isabelle, described below.

2.3.1 Automatic Systems

The automatic systems strive to prove theorems without intervention from humans. As we now know theorem proving is an undecidable problem; this means these systems have to employ strategies beyond pure logical reasoning. For this reason, automatic theorem provers have been among the most prominent areas of application of artificial intelligence research.

Boyer-Moore

A classic system, made by Robert S. Boyer and J. Strother Moore, and described in e.g. [BM79]. Automatic in execution, but in the authors' opinion very much dependent on the introduction of helpful lemmas aforehand by the user. The lemmas must themselves be proved, so one prepares the system for a complex proof by making it prove simpler lemmas, each successively relying on those proved previously, and starting with ones the system manages to prove from its built-in axioms. Because of this, Boyer and Moore further write, the user has to be fairly adept at logic to find the lemmas that need to be proven by the system. The Boyer-Moore prover therefore in effect does much the same as the ABEL system—it takes care of the tedious, completely mechanical parts of proof construction, and it guarantees that the generated proof do not contain errors.

Argonne/Otter

Groups at the Argonne National Laboratory started working with automatic theorem provers in the early 1960s, and research there is still going strong, now with the system called *Otter*. Otter is an automated deductive prover designed to prove theorems in first-order logic with equality. Two other systems are currently also developed at Argonne; *EQP*, which searches for equational proofs, and *MACE*, which searches for models and counter-examples. All can be tested online using a system called *Son of BirdBrain*, see [arg].

Larry Wos and William McCune have been the primary researchers in the Argonne theorem proving efforts, McCune being the primary developer behind Otter. Further information on the history of Otter and the theorem proving effort at Argonne can be found in Rusty Lusk's paper [Lus92], and on the web at [arg].

SPIKE

SPIKE is an inductive prover, and seemingly rather basic at that. It uses induction and rewriting as methods, but only uses implicitly quantification of the variables, and will therefore quite easily diverge. [Wal96] exemplifies divergence by showing SPIKE alone trying to prove the theorem $S(x+x) =$

$S(x)+x$. (The axiom for addition in SPIKE is $x+S(y) = S(x+y)$.) Trying to prove this theorem SPIKE repeatedly applies induction to x, but is unable to simplify the generated expressions by rewriting.

The divergence critic described in [Wal96] discovers this divergence, and guesses lemmas that hopefully are both helpful and may be proved without themselves causing divergence. This divergence critic functions remarkably well, and has been applied with success also to the Boyer-Moore system. However, divergence is not a problem for the ABEL system, as induction is (and should be) only allowed to be applied once to a variable in any one proof branch.

2.3.2 Interactive Systems

With the lack of success for automatic theorem provers, research shifted to interactive modes of operation, where the system assists the user in constructing the proof. Some systems are little more than proof checkers that make sure all proof steps taken by the user are sound, while others, like PVS, provide the user with a few high-level commands that do most of the work.

PVS

Like the ABEL system, PVS[1] is an interactive system; it is the system with the greatest resemblance to ours that I will discuss. A difference to the ABEL system is that the language used in PVS is a pure specification language, while ABEL is also meant as a language for implementation. A reference for the PVS language may be found in [SORSC99a]. PVS is developed by SRI International.

The PVS prover, described in [SORSC99b], has an interface somewhat similar to the ABEL system, with commands for logical reasoning and induction. Rewriting is done automatically. Where PVS differs the most from our system is in the focus on high-level strategies for practical proving. PVS combines all the commands for logical reasoning (similar to the ABEL system's BPC rules) into the two strategies split and flatten, and the powerful strategy grind, which incorporates split and flatten. split contains all the rules that cause splitting of the sequent, while flatten contains those that do not cause splitting. Brevik discusses this in [Bre98] as compared to the functioning of his prover, and has implemented split and flatten (in [Bre98] pp. 131-132) as ABEL strategies.

PVS has been used in practice in the specification and partial verification of the Rockwell-Collins AAMP5 processor design [MS95], and in the verification of a SRT divider [RSS99].

[1] URL: http://pvs.csl.sri.com/

Isabelle

Isabelle is the name of the system currently being developed by Lawrence C. Paulson and others at the University of Cambridge and the Technical University of München. Isabelle's ancestry goes back to LCF, an automatic prover developed by Paulson in the 1970s; incidentally, the system where the ML programming language originated (as a metalanguage).

Although it has a heritage from automatic provers, Isabelle is now, in similarity to PVS and the ABEL system, interactive, and incorporates several proof techniques.

Isabelle has been used for proving properties of the Kerberos authentication system.

2.4 How do Humans Construct Proofs?

One very interesting question, especially in the construction of automatic theorem provers and research into artificial intelligence is, how do we humans construct proofs? Do we have techniques that somehow defy the undecidability of proof construction, or do we just use experience and the capability of the human brain—which still vastly outperforms computers? This last issue is of course one of the most central and most difficult questions of artificial intelligence research, pondered since Turing first came up with his model for a universal computing machine, so I will not try to answer it here.

It is worth a note, though, that one of the ways humans seem to outperform computers the most, is in the ability to draw on previous experience—such as in choosing which of a number of proof techniques applicable at some point in a proof that seems most promising. My personal opinion is that this alone may justify at least a large part of the difference in the proof construction capability between humans and computers.

Chapter 3

A User's Guide to the ABEL proof system

The ABEL system is divided into several parts: the base system, which among other things handles the command interpretation and translation of ABEL program code into internal structures; the type checker; the rewriter; the prover. In using the system, one mostly comes in contact with the base system's toplevel and parser, and the prover; the rewriter has its user interface through the commands rewrite, xrewrite and rewruleset, and type checking is done automatically.

3.1 The Base System Commands

context: Loads a module into the current context. Standard modules include Int, Seq, Set and Group.

eval: Evaluates an expression in the host language.

help: Provides simple help on commands.

print: Displays useful information; currently the number of rewrite steps, the rewrite rules and the latest rewrite steps are supported.

prove: Invokes the prover module to prove one or more expressions given as arguments.

rewrite: Invokes the rewrite module to rewrite an expression given as an argument.

rewruleset: Manages the rewrite ruleset; with this command one can list, add and delete rewrite rules.

set: Sets a system variable; when not given an argument lists all set variables.

quit, exit, bye, return: Quits to the next level up, or if at toplevel, quits the ABEL system.

3.2 The Prover

3.2.1 The Commands

The prover introduces a number of commands in addition to what is accessible in the base system. The base system commands (with the obvious exception of prove) work as described above, where not noted otherwise below.

assert: Asserts to the system that a subexpression is true. This can be done if one does not want to continue a proof branch, e.g. if one sees that the current subexpression is trivially true, but the system seems unable to prove it.

disclaim: Reverts asserts; without arguments lists all asserted subexpressions.

induct: Invokes the induction module; takes as compulsory argument the part of the consequent to be used.

next: Changes to the next expression.

open: Opens a strategy file.

postpone: Postpones the current proof branch. Using this command the user can cycle through all the proof branches.

prev: Changes to the previous expression (the opposite of next).

printproof: Prints the proof steps together with what rules were used. Applications of strategies are not included in the printout, but rather the rules the strategies resulted in.

prune: "Prunes" off a whole proof branch. If a proof branch has gone in the wrong direction, instead of performing multiple undo commands, using this command on can remove the whole branch and start anew at the last branching point.

rewrite: Invokes the rewriter on the current subexpression. When invoked from the prover (as opposed to the toplevel), this command does not take an expression as argument, but works (as all prover commands) on the current expression.

xrewrite: Invokes an improved rewriter (described in section 4.1).

ruleset: Lists, adds or deletes rulesets.

strategy: Stores a new strategy in a file.

undo: Reverts the last action.

view: Lists the sequence of successfully applied rules.

3.2.2 The Rules

cut: As detailed in [Bre98] p. 111, Brevik has implemented this rule to facilitate a form of model checking.

split: A strategy consisting of all the constructive BPC rules that causes splitting in the sequent: tand, aor, aimpl, tif and aif.

flatten: A strategy of the constructive BPC rules that does not cause splitting in the sequent: aand, tor, timpl, anot, tnot, teqv, aeqv, aexist and tall.

bpc: The bpc strategy is implemented as a combination of first the flatten strategy, and then split, repeated until neither is successful.

3.2.3 The Command Language

multi: Applies a command on every applicable part of the current sequent.

repeat: Repeats a command list until the proof is completed or the command list fails. Takes an optional integer argument for how many times to repeat.

(): Parentheses put around a list of proof rules or commands (separated by comma) executes the rules in sequence.

[]: Square brackets put around a list of proof rules or commands (separated by comma) executes the rules in sequence until one succeeds.

3.3 The ABEL Language

While the imperative parts of ABEL clearly shows its ancestry in the ALGOL family, some of the applicative constructs for type generation are syntactically alike to the equivalent constructs in ML and similar functional languages. I will not describe the full language, as much of it should be obvious. However, there are some parts which merit some explanation. I will use figure 3.1 as example code for this.

```
Tree ==
module
  include Int

  typevar T

  type Tree{T} by ETree, NETree

  func nil : ETree{T}
  func leaf : T ⟶ NETree{T}
  func tree : Tree{T} * Tree{T} ⟶ Tree{T}

  oneone genbas Tree == nil, leaf, tree

  func del : Tree{T} * T ⟶ Tree{T}
  func ^_sub_^ : Tree{T} * Tree{T} ⟶ Bool

  def del(t, x) ==
    case t of nil ⟶ nil
            | leaf(y) ⟶ if x = y then nil else t fi
            | tree(u, v) ⟶ tree(del(u, x), del(v, x)) fo

  def s _sub_ t == (s = t) ∨
                   case t of nil ⟶ false
                           | leaf(x) ⟶ false
                           | tree(u, v) ⟶ s _sub_ u ∨ s _sub_ v fo

endmodule
```

Figure 3.1: The Tree module

3.3.1 The Module

The facility in ABEL for modularisation is the module construct. Its syntax is, as we see in figure 3.1, first the module's name, then ==, and then the module block, with module as start tag and endmodule as end.

This module uses functions from the Int module, hence it uses the include declaration to include this module. It then proceeds to introduce the *Tree* type, by declaring its generators (nil, leaf and tree). The generator functions are only declared, not defined. The oneone genbas declaration declares these functions as the *Tree* type's generator basis.

After this comes the rest of the functions of the module. Ordinary functions are introduced by first declaring their type signature with func, and then their implementation with def.

3.3.2 case

The case construct is used in almost all generator and observer functions. It bears a syntactic resemblance to the ML case construct, but in ABEL it dispatches only on type generators. Let us as an example look at the del function of figure 3.1; here there are three branches, one for each of the generators nil, leaf and tree.

3.3.3 Infix Function Declaration

Another thing to note in figure 3.1 is the declaration of the infix function _sub_; in the func statement, ^ declares the placement of the arguments.

3.4 A Sample Proof

Now let us see how a simple proof session might be conducted. We use the simple theorem $\forall(x, y : \text{Nat})x + Sy = S(x + y)$ for illustration.

First we instruct the system to prove this theorem:

```
ABEL> prove forall(x,y:Nat) x + S y = S (x + y)
```

The system acknowledges this by printing:

```
1:

 True (empty)
|-----
# 1) forall(x:Nat,y:Nat) x + S y = S (x + y)
```

Now, induction seems a plausible path forward, so we use the induct command, and give as argument '1', as this is the number of the (only) part of the consequent to be used.

```
PROVER> induct 1
This yields 2 subgoals:

1.1:

 True (empty)
|-----
# 1) forall(y:Nat) 0 + S y = S (0 + y)


1.2:

#-1) forall(y:Nat) x + S y = S (x + y)
|-----
# 1) forall(y:Nat) S x + S y = S ((S x) + y)
```

The induction generates these two sequents, 1.1 and 1.2, as new subgoals. The system chooses 1.1 as the new current sequent, and announce this by printing it again:

```
1.1:

 True (empty)
|-----
# 1) forall(y:Nat) 0 + S y = S (0 + y)
```

After this we only need two rewrites to complete the proof:

```
PROVER> xrewrite

1.1:

 True (empty)
|-----
# 1) true
Which is trivially true!



Changing current to '1.2'.

1.2:

#-1) forall(y:Nat) x + S y = S (x + y)
|-----
# 1) forall(y:Nat) S x + S y = S ((S x) + y)

PROVER> xrewrite
```

```
1.2:

#-1) forall(y:Nat) x + S y = S (x + y)
|-----
# 1) true
Which is trivially true!

The proof consists of 5 nodes.


Q.E.D.

This concludes the proof of this expression.
You might store a strategy for the proof by using the
'strategy' command.
('exit' terminates the prover.)
```

To see the whole proof we can use the printproof command:

```
PROVER> printproof
1)  ||-- forall(x:Nat,y:Nat) x + S y = S (x + y) (induct  2,4)
2)  ||-- forall(y:Nat) 0 + S y = S (0 + y) (xrewrite  3)
3)  ||-- true (TRIV)
4)  forall(y:Nat) x + S y = S (x + y)
 ||-- forall(y:Nat) S x + S y = S ((S x) + y) (xrewrite  5)
5)  forall(y:Nat) x + S y = S (x + y) ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

3.5 A Larger Proof

Now we will see how to do a slightly larger proof, which use the *Tree* module in figure 3.1. The proof involves induction over the *Tree* type, which here generates three branches—one for each of the generators, nil, leaf and tree. The theorem we want to prove is:

$$\forall t : \text{Tree}, x : \text{T} \cdot \text{del}(\text{del}(t, x), x) = \text{del}(t, x)$$

We start by introducing the module and the theorem:

```
ABEL> context Tree
Parsing: /hom/martint/src/hovedfag/Abel/AbelSrc/Tree.abl
Checking: /hom/martint/src/hovedfag/Abel/AbelSrc/Tree.abl
Added rewrite rules for Tree
ABEL> prove forall(t:Tree{T},x:T) del(del(t, x), x) = del(t, x)
```

```
1:

 True (empty)
|-----
# 1) forall(t:Tree{T},x:T) del (del (t,x),x) = del (t,x)
```

Now we have to choose what command to use. We try the rewriter, but find that it cannot do anything, so we try bpc instead:

```
PROVER> xrewrite
Gives no changes.

PROVER> bpc

1:

 True (empty)
|-----
# 1) del (del (t,x),x) = del (t,x)

PROVER> xrewrite
Gives no changes.
```

Here, bpc removed the universal quantifier, but the rewriter could not change the resulting formula; and, obviously, neither can induction nor the non-constructive BPC-rules. We therefore go back and try another route.

```
PROVER> undo
Undoing the last proof-step.


1:

 True (empty)
|-----
# 1) forall(t:Tree{T},x:T) del (del (t,x),x) = del (t,x)

PROVER> induct 1
This yields 3 subgoals:

1.1:

 True (empty)
|-----
# 1) forall(x:T) del (del (nil,x),x) = del (nil,x)


1.2:
```

```
 True (empty)
|-----
# 1) forall(x:T) del (del (leaf t'el,x),x) = del (leaf t'el,x)


1.3:

#-1) forall(x:T) del (del (t,x),x) = del (t,x)
|-----
# 1) forall(x:T) del (del (tree (t,t),x),x) = del (tree (t,t),x)
```

Here in 1.2 we see how the induction subsystem generates fresh variable names: it appends "'el" to the original variable name.

```
1.1:

 True (empty)
|-----
# 1) forall(x:T) del (del (nil,x),x) = del (nil,x)

PROVER> xrewrite

1.1:

 True (empty)
|-----
# 1) true
Which is trivially true!



Changing current to '1.2'.

1.2:

 True (empty)
|-----
# 1) forall(x:T) del (del (leaf t'el,x),x) = del (leaf t'el,x)
```

It seems induction was indeed a better choice than the `bpc` application we had to revert. Now we have to prove the second branch from the induction:

```
PROVER> xrewrite

1.2:

 True (empty)
|-----
```

```
# 1) forall(x:T)
     if x = t'el
       then x = t'el
       else if x = t'el
              then x = t'el
              else if x = t'el then false else true fi
            fi
     fi

PROVER> bpc

1.2.2.2.2:

#-1) ~ (x = t'el)
|-----
# 1) true, # 2) x = t'el
Which is trivially true!



Changing current to '1.3'.

1.3:

#-1) forall(x:T) del (del (t,x),x) = del (t,x)
|-----
# 1) forall(x:T) del (del (tree (t,t),x),x) = del (tree (t,t),x)
```

The tif rule took care of that, so now we only have the induction step left, which the rewriter proves:

```
PROVER> xrewrite

1.3:

#-1) forall(x:T) del (del (t,x),x) = del (t,x)
|-----
# 1) true
Which is trivially true!

The proof consists of 22 nodes.


Q.E.D.
```

printproof prints out all the rules used in the proof, unraveling the rules applied in the bpc applications:

```
PROVER> printproof
```

```
1)  ||-- forall(t:Tree{T},x:T) del (del (t,x),x) = del (t,x)
 (induct  2,4,21)
2)  ||-- forall(x:T) del (del (nil,x),x) = del (nil,x)
 (xrewrite  3)
3)  ||-- true (TRIV)
4)  ||-- forall(x:T) del (del (leaf t'el,x),x) = del (leaf t'el,x)
 (xrewrite  5)
5)  ||-- forall(x:T) if x = t'el
               then x = t'el
               else if x = t'el
                      then x = t'el
                      else if x = t'el then false else true fi
                    fi
             fi (tall  6)
6)  ||-- if x = t'el then x = t'el
   else if x = t'el then x = t'el
   else if x = t'el then false else true fi fi
 fi (tif  7,9)
7)  ||-- x = t'el => x = t'el (timpl  8)
8)  x = t'el ||-- x = t'el (TRIV)
9)  ||-- ~ (x = t'el) =>
 if x = t'el then x = t'el
 else if x = t'el then false else true fi fi (timpl  10)
10)  ~ (x = t'el) ||-- if x = t'el then x = t'el
 else if x = t'el then false else true fi fi (anot  11)
11)  ||-- if x = t'el then x = t'el
 else if x = t'el then false else true fi fi,
 x = t'el (tif  12,14)
12)  ||-- x = t'el => x = t'el,  x = t'el (timpl  13)
13)  x = t'el ||-- x = t'el (TRIV)
14)  ||-- ~ (x = t'el) => if x = t'el then false
 else true fi,  x = t'el (timpl  15)
15)  ~ (x = t'el) ||-- if x = t'el then false
 else true fi,  x = t'el (anot  16)
16)  ||-- if x = t'el then false else true fi,  x = t'el
 (tif  17,19)
17)  ||-- x = t'el => false,  x = t'el (timpl  18)
18)  x = t'el ||-- x = t'el (TRIV)
19)  ||-- ~ (x = t'el) => true,  x = t'el (timpl  20)
20)  ~ (x = t'el) ||-- true,  x = t'el (TRIV)
21)  forall(x:T) del (del (t,x),x) = del (t,x)
 ||-- forall(x:T) del (del (tree (t,t),x),x) = del (tree (t,t),x)
 (xrewrite  22)
22)  forall(x:T) del (del (t,x),x) = del (t,x) ||-- true (TRIV)
The proof consists of 22 nodes.


Q.E.D.
```

As we see the proof was fairly straightforward, but still needed some

user guidance in choosing whether to treat the universal quantifier by induction or BPC.

Chapter 4

A First Test of the ABEL System

As a starting point for this thesis, I tested the ABEL system's prover module with a list of theorems found in [Wal96] (p. 226). That article describes the functioning of a "divergence critic"—a system that proposes (hopefully useful) lemmas when the proof construction of its accompanying theorem prover diverges—for SPIKE ([BKR92]), an automatic, inductive theorem prover (as described in section 2.3.1). The mentioned list lists theorems SPIKE managed to prove when augmented by the critic, and the lemmas the critic proposed that made SPIKE able to complete the proofs.

4.1 xrewrite

As described in this chapter, none of the theorems pose serious trouble for the ABEL system. However, this was not quite the case when I first did these tests; the introduction of the xrewrite command (written by Jo Totland), and its ability to construct rewrite rules from the antecedent of the current sequent, thereby providing instantiation capabilities that were not previously present in the system, is the single reason why the system is now able to complete all the proofs. I will give a few examples of where this made a difference in the following.

4.2 Implementation

The theorems to be tested uses a number of functions; I have implemented those functions not included in the ABEL system's standard *Int* and *Seq* modules (included in appendix A). To use these functions, I encapsulated them in a module on a file suitably situated in the ABEL system's file hierarchy and used the context command in the system's toplevel to load them. The module can be found in subsection 4.3.1. The functions use the *Int* and *Seq* modules; the module also introduces the general type variable *T*.

A number of the functions that are used in the theorems are described in [Wal96] by case-free axioms; these I have implemented as prescribed, by translating into case statements in ABEL (which are described in section 3.3. The others I implemented as best I could, keeping proof construction (and especially induction) in mind. The only functions I have not implemented myself of those used in the theorems, are those already present in the standard modules: the integer successor function *S* from the *Int* module; and the standard sequence operators $\vdash$ (right concatenation), $\dashv$ (left concatenation), $\vdash\!\dashv$ (concatenation of two sequences, @ is used in [Wal96]), and # (length) from the *Seq* module.

4.3 Source Code

Here follows the ABEL code I used in these tests, the functions and the theorems.

4.3.1 Functions

Here are presented the functions I needed to implement for the tests, that is, the functions used in [Wal96] that is not already implemented in the ABEL system. I made a module to hold the functions and a general type variable T; the "func g : T" and oneone genbas T == g" statements below are ad-hoc workarounds to make the module syntactically right, and has no practical effect. I also used the standard ABEL modules Int and Seq, which can be found in appendix A.

Functions in ABEL is defined by first stating their type profile with the func keyword, and then their implementation with the def keyword. All the following functions are implemented by type induction with the case construct, which in syntax resembles the case construct found in e.g. ML. It matches an expression against the expressions of the different branches sequentially, choosing the branch for which it first finds a match.

```
Essay ==
module
  include Int
  include Seq

  type T
  func g : T

  oneone genbas T == g
```

```
  func dbl : Nat ⟶ Nat (* double the argument *)
  func even : Nat ⟶ Bool (* is the argument even? *)
  func odd : Nat ⟶ Bool (* is the argument odd? *)
  func even' : Nat ⟶ Bool (* another implementation of even *)
  func odd' : Nat ⟶ Bool (* another implementation of odd *)
  func half : Nat ⟶ Nat (* half the argument *)
  func rot : Nat * Seq{T} ⟶ Seq{T} (* rotate second argument by first *)
  func rev : Seq{T} ⟶ Seq{T} (* reverse the argument *)
  func qrev : Seq{T} * Seq{T} ⟶ Seq{T} (* reverse first argument at
                                          the end of the second *)
  func nth : Nat * Seq{T} ⟶ Seq{T} (* the nth argument of the seq *)
  func isort : Seq{Int} ⟶ Seq{Int} (* sort by insertion sort *)
  func sorted : Seq{Int} ⟶ Bool (* is the argument sorted? *)

  def dbl(x) == case x of Z ⟶ 0 | S(x) ⟶ S(S(dbl(x))) fo
  def even(x) == case x of Z ⟶ true | S(Z) ⟶ false | S(S(x)) ⟶ even(x) fo
  def odd(x) == case x of Z ⟶ false | S(Z) ⟶ true | S(S(x)) ⟶ odd(x) fo
  def even'(x) == case x of Z ⟶ true | S(x) ⟶ odd'(x) fo
  def odd'(x) == case x of Z ⟶ false | S(x) ⟶ even'(x) fo
  def half(x) == case x of Z ⟶ 0 | S(x) ⟶
         case x of Z ⟶ 0 | S(x) ⟶ S(half(x)) fo fo
  def rot(n, s) == case n of Z ⟶ s | S(n') ⟶
         case s of e ⟶ e |  s'⊢x ⟶ rot(n', x−|s') fo fo

  def rev(s) == case s of e ⟶ e | s'⊢x ⟶ x −| rev(s') fo
  def qrev(s, r) == case s of e ⟶ r | s⊢x ⟶ qrev(s, r⊢x) fo
  def nth(i, s) == case i of Z ⟶ s | S(i) ⟶
                          case s of e ⟶ e | s⊢x ⟶ nth(i, s) fo fo

  func insert : Int * Seq{Int} ⟶ Seq{Int}
  def insert(x, s) ==
    case s of e ⟶ e⊢x
              | s'⊢y ⟶ if x<y then s⊢x else insert(x, s')⊢y fi fo

  def isort(s) == case s of e ⟶ e | s⊢x ⟶ insert(x, isort(s)) fo
  def sorted(s) ==
    case s of e ⟶ true | e⊢x ⟶ true | s⊢x ⟶
      case s of s'⊢y ⟶ if y<x then sorted(s) else false fi
                | _ ⟶ false fo fo
endmodule
```

4.3.2 Theorems

Here are the theorems from [Wal96], translated to ABEL syntax. I have, as mentioned in the introduction, reworked the theorems which include sequence operations to accommodate for the reversal of sequence concatenation, in that right concatenation is the sequence generator in ABEL, as opposed to left concatenation in SPIKE.

1. $\forall$(x:Nat) S(x) + x = S(x + x)

2. $\forall$(x:Nat) dbl(x) = x + x $\leq>$ dbl(0) = 0

3. $\forall$(x:Nat) dbl(S(x)) = S(S(dbl(x)))

4. $\forall$(x,y:Seq{T}) #(x⊢|y) = #(y⊢|x)

5. $\forall$(x,y:Seq{T}) #(x⊢|y) = #x + #y

6. $\forall$(x:Seq{T}) #(x⊢|x) = dbl(#x)

7. $\forall$(x:Nat) even(x+x)

8. $\forall$(x:Nat) odd(S(x)+x)

9. $\forall$(x:Nat) even'(x+x)

10. $\forall$(x:Nat) odd'(S(x)+x)

11. $\forall$(x:Nat) even'(x) $\Rightarrow$ half(x) + half(x) = x

12. $\forall$(x:Nat) half(x+x) = x

13. $\forall$(x:Nat) half(S(x)+x) = x

14. $\forall$(x:Seq{T}) rot(#x, x) = x

15. ∀(x:Seq{T}) #(rot(#x, x)) = #x

16. ∀(x:Seq{T}, y:T) rot (# x,(e ⊢ y) ⊢| x) = rot (# x, x) ⊢ y

17. ∀(x:Seq{T}) #(rev(x)) = #x

18. ∀(x:Seq{T}) rev(rev(x)) = x

19. ∀(x:Seq{T},y:T) rev (e ⊢ y ⊢| rev x) = x ⊢ y

20. ∀(x:Seq{T},y:T) rev (e ⊢ y ⊢| rev x) = rev (rev x) ⊢ y

21. ∀(x,y:Seq{T}) # (rev (x ⊢| y)) = # x + # y

22. ∀(x:Seq{T}) #(qrev(x, e)) = #x

23. ∀(x,y:Seq{T}) qrev(x, y) = y ⊢| rev(x)

24. ∀(x,y:Seq{T}) #(qrev(x, y)) = #x + #y

25. ∀(x:Seq{T}) qrev(qrev(x, e), e) = x

26. ∀(x:Seq{T}) rev(qrev(x, e)) = x

27. ∀(x:Seq{T}) qrev(rev(x), e) = x

28. ∀(i,j:Nat, x:Seq{T}) nth(i, nth(j, x)) = nth(j, nth(i, x))

29. ∀(i,j,k:Nat, x:Seq{T}) nth(i, nth(j, nth(k, x))) = nth(k, nth(j, nth(i, x)))

30. ∀(x:Seq{Int}) #(isort(x)) = #x

31. ∀(x:Seq{Int}) sorted(isort(x))

4.4 Results

SPIKE, the proof system that [Wal96] uses, is an automatic theorem prover, while the proof module of the ABEL system is interactive—that is, part of what the system in [Wal96] does, is left to the user by design in the ABEL system. This means that our system does not complete all the proofs without help, but in some of the proofs I experienced that our proof system have much less need than SPIKE (without the divergence critic) for help, be it introduction of new lemmas or user intervention in choosing between using induction or BPC rules.

One of the problems I encountered with the ABEL system is that to apply induction to a variable the variable has to be bound in a universal quantifier. The BPC rule $T\forall$, which is included in the bpc strategy, may remove such quantifiers in an application of bpc, making it necessary to investigate which BPC rules were used with printproof, revert the bpc application with undo, before reapplying the BPC rules except tall and at last being able to use induction. If there was a generalisation facility in the system, the user would not need to do that rather tedious sequence of manual operations.

I have not included the whole proof sessions, just the printout from the printproof command; as this printout often contains too long lines I have reformatted it somewhat to fit the page. printproof prints the proof steps starting with the theorem to be proved, and includes at each steps the rule used and what sequents resulted from this application. I will use the first proof as an example for a more detailed explanation below.

4.4.1 $S(x) + x = S(x + x)$

```
1)  ||-- forall(x:Nat) S x + x = S (x + x) (induct  2,4)
2)  ||-- 1 + 0 = S (0 + 0) (rewrite  3)
3)  ||-- true (TRIV)
4)  S x + x = S (x + x)
    ||-- S (S x) + S x = S ((S x) + S x) (rewrite  5)
5)  S x + x = S (x + x) ||-- S x + x = S (x + x) (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

Here we see that the first rule used was induct, which produced sequents 2 (the base step) and 4 (the induction step). The rewriter managed to rewrite each of these into trivial sequents (3 and 5). As there was then no non-trivial sequents left, the proof was declared successful. This should be seen as a quite straightforward proof; all the user needs to do is try rewrite repeatedly, using induct when rewrite does not give any changes to the sequent.

It is worth to note that addition in SPIKE is defined by $x + S(x) = S(x + x)$, while in ABEL it is defined the other way around, i.e. $S(x)+x = S(x+x)$. However, the ABEL system's proof for $x + S(x) = S(x + x)$ is similar to the proof above.

4.4.2 $(x : Nat)dbl(x) = x + x <=> dbl(0) = 0$

```
1)  ||-- forall(x:Nat) dbl x = x + x <=> dbl 0 = 0 (teqv  2)
2)  ||-- ((forall(x:Nat) dbl x = x + x) => dbl 0 = 0) /\
 (dbl 0 = 0 => (forall(x:Nat) dbl x = x + x)) (tand  3,6)
3)  ||-- (forall(x:Nat) dbl x = x + x) => dbl 0 = 0 (timpl  4)
4)  forall(x:Nat) dbl x = x + x ||-- dbl 0 = 0 (rewrite  5)
5)  forall(x:Nat) dbl x = x + x ||-- true (TRIV)
6)  ||-- dbl 0 = 0 => (forall(x:Nat) dbl x = x + x) (rewrite  7)
7)  ||-- forall(x:Nat) dbl x = x + x (induct  8,10)
8)  ||-- dbl 0 = 0 + 0 (rewrite  9)
9)  ||-- true (TRIV)
10)  dbl x = x + x ||-- dbl (S x) = S x + S x (rewrite  11)
11)  dbl x = x + x ||-- dbl x = x + x (TRIV)
The proof consists of 11 nodes.


Q.E.D.
```

As we see, ABEL manages to also construct this proof with few problems. One thing to be noticed is that induction does not work at step 1; the bpc strategy had to be applied first to take care of the equivalence. The command sequence used here was (bpc, rewrite, rewrite, induct 1, rewrite, rewrite). The printproof command prints out all the rules applied in a strategy application rather than just the strategy.

4.4.3 $dbl(S(x)) = S(S(dbl(x)))$

```
1)  ||-- forall(x:Nat) dbl (S x) =
           S (S (dbl x)) (rewrite  2)
2)  ||-- true (TRIV)
The proof consists of 2 nodes.


Q.E.D.
```

The rewriter proves this theorem in one step. The simple proof of this theorem is not very surprising, as this is precisely the way that I have implemented dbl.

4.4.4 $len(x@y) = len(y@x)$

```
1)  ||-- forall(x:Seq{T},y:Seq{T}) # (x |-| y) = # (y |-| x)
```

```
 (induct  2,8)
2)  ||-- forall(y:Seq{T}) # (e |-| y) = # (y |-| e) (xrewrite  3)
3)  ||-- forall(y:Seq{T}) # (e |-| y) = # y (induct  4,6)
4)  ||-- # (e |-| e) = # e (xrewrite  5)
5)  ||-- true (TRIV)
6)  # (e |-| y) = # y ||-- # (e |-| (y |- y'el)) = # (y |- y'el)
 (xrewrite  7)
7)  # (e |-| y) = # y ||-- true (TRIV)
8)  forall(y:Seq{T}) # (x |-| y) = # (y |-| x)
 ||-- forall(y:Seq{T}) # ((x |- x'el) |-| y) = # (y |-| (x |- x'el))
 (xrewrite  9)
9)  forall(y:Seq{T}) # (x |-| y) = # (y |-| x) ||-- true (TRIV)
The proof consists of 9 nodes.


Q.E.D.
```

The preceding theorems were proved with only use of Korshavn and Midtskogen's rewriter (rewrite), but this proof I am not able to complete without the improved version (xrewrite), as the following proof attempt shows:

```
1)  ||-- forall(x:Seq{T},y:Seq{T}) # (x |-| y) =
 # (y |-| x) (induct  2,8)
2)  ||-- forall(y:Seq{T}) # (e |-| y) = # (y |-| e)
 (rewrite  3)
3)  ||-- forall(y:Seq{T}) # (e |-| y) = # y (induct  4,6)
4)  ||-- # (e |-| e) = # e (rewrite  5)
5)  ||-- true (TRIV)
6)  # (e |-| y) = # y ||-- # (e |-| (y |- y'el)) =
 # (y |- y'el) (rewrite  7)
7)  # (e |-| y) = # y ||-- # (e |-| y) = # y (TRIV)
8)  forall(y:Seq{T}) # (x |-| y) = # (y |-| x)
 ||-- forall(y:Seq{T}) # ((x |- x'el) |-| y) =
 # (y |-| (x |- x'el)) (rewrite  9)
9)  forall(y:Seq{T}) # (x |-| y) = # (y |-| x)
 ||-- forall(y:Seq{T}) # ((x |- x'el) |-| y) =
 S (# (y |-| x)) (induct  10,14)
10)  forall(y:Seq{T}) # (x |-| y) = # (y |-| x)
 ||-- # ((x |- x'el) |-| e) = S (# (e |-| x)) (rewrite  11)
11)  forall(y:Seq{T}) # (x |-| y) = # (y |-| x)
 ||-- # x = # (e |-| x) (aall  12)
12)  # (x |-| e) = # (e |-| x),
 forall(y:Seq{T}) # (x |-| y) = # (y |-| x)
 ||-- # x = # (e |-| x) (rewrite  13)
13)  # x = # (e |-| x),
 forall(y:Seq{T}) # (x |-| y) = # (y |-| x)
 ||-- # x = # (e |-| x) (TRIV)
14)  forall(y:Seq{T}) # (x |-| y) = # (y |-| x),
```

```
 # ((x |- x'el) |-| y) = S (# (y |-| x))
 ||-- # ((x |- x'el) |-| (y |- y'el)) =
 S (# ((y |- y'el) |-| x)) (rewrite  15)
15)  forall(y:Seq{T}) # (x |-| y) = # (y |-| x),
 # ((x |- x'el) |-| y) = S (# (y |-| x))
 ||-- # ((x |- x'el) |-| y) = # ((y |- y'el) |-| x)
 (aall  16)
16)  # ((x |- x'el) |-| y) = S (# (y |-| x)),
 # (x |-| (y |- y'el)) = # ((y |- y'el) |-| x),
 forall(y:Seq{T}) # (x |-| y) = # (y |-| x)
 ||-- # ((x |- x'el) |-| y) = # ((y |- y'el) |-| x)
 (rewrite  17)
17)  # ((x |- x'el) |-| y) = S (# (y |-| x)),
 S (# (x |-| y)) = # ((y |- y'el) |-| x),
 forall(y:Seq{T}) # (x |-| y) = # (y |-| x)
 ||-- # ((x |- x'el) |-| y) = # ((y |- y'el) |-| x)
```

As we see, the two proof attempts diverge at step 9, where xrewrite manages to rewrite the consequent to true by constructing a rewrite rule from the antecedent; in other words, the first proof attempt uses the induction hypothesis.

4.4.5 $len(x@y) = len(x) + len(y)$

```
1)  ||-- forall(x:Seq{T},y:Seq{T}) # (x |-| y) = # x + # y
 (induct  2,8)
2)  ||-- forall(y:Seq{T}) # (e |-| y) = # e + # y (xrewrite  3)
3)  ||-- forall(y:Seq{T}) # (e |-| y) = # y (induct  4,6)
4)  ||-- # (e |-| e) = # e (xrewrite  5)
5)  ||-- true (TRIV)
6)  # (e |-| y) = # y ||-- # (e |-| (y |- y'el)) =
 # (y |- y'el) (xrewrite  7)
7)  # (e |-| y) = # y ||-- true (TRIV)
8)  forall(y:Seq{T}) # (x |-| y) = # x + # y
 ||-- forall(y:Seq{T}) # ((x |- x'el) |-| y) =
 # (x |- x'el) + # y (xrewrite  9)
9)  forall(y:Seq{T}) # (x |-| y) = # x + # y ||-- true (TRIV)
The proof consists of 9 nodes.


Q.E.D.
```

Here again the xrewrite command makes the construction of this proof relatively simple, while as shown below, with only rewrite the proof both is longer, and flounders on the last step. It should be noted that xrewrite of sequent 12 below leads to completion of the proof.

```
1)  ||-- forall(x:Seq{T},y:Seq{T})
```

```
         # (x |-| y) = # x + # y (induct  2,8)
2)  ||-- forall(y:Seq{T})
         # (e |-| y) = # e + # y (rewrite  3)
3)  ||-- forall(y:Seq{T})
         # (e |-| y) = # y (induct  4,6)
4)  ||-- # (e |-| e) = # e (rewrite  5)
5)  ||-- true (TRIV)
6)  # (e |-| y) = # y
    ||-- # (e |-| (y |- y'el)) =
         # (y |- y'el) (rewrite  7)
7)  # (e |-| y) = # y ||-- # (e |-| y) = # y (TRIV)
8)  forall(y:Seq{T}) # (x |-| y) = # x + # y
    ||-- forall(y:Seq{T}) # ((x |- x'el) |-| y) =
         # (x |- x'el) + # y (rewrite  9)
9)  forall(y:Seq{T}) # (x |-| y) = # x + # y
    ||-- forall(y:Seq{T}) # ((x |- x'el) |-| y) =
         S (# x) + # y (induct  10,12)
10)  forall(y:Seq{T}) # (x |-| y) = # x + # y
     ||-- # ((x |- x'el) |-| e) =
          S (# x) + # e (rewrite  11)
11)  forall(y:Seq{T}) # (x |-| y) = # x + # y
     ||-- true (TRIV)
12)  forall(y:Seq{T}) # (x |-| y) = # x + # y;
     # ((x |- x'el) |-| y) = S (# x) + # y
     ||-- # ((x |- x'el) |-| (y |- y'el)) =
          S (# x) + # (y |- y'el) (rewrite  13)
13)  forall(y:Seq{T}) # (x |-| y) = # x + # y;
     # ((x |- x'el) |-| y) = S (# x) + # y
     ||-- # ((x |- x'el) |-| y) = S ((# x) + # y)
```

4.4.6 $len(x@x) = dbl(len(x))$

```
1)  ||-- forall(x:Seq{T}) # (x |-| x) = dbl (# x) (induct  2,4)
2)  ||-- # (e |-| e) = dbl (# e) (rewrite  3)
3)  ||-- true (TRIV)
4)  # (x |-| x) = dbl (# x) ||-- # ((x |- x'el) |-| (x |- x'el)) =
 dbl (# (x |- x'el)) (xrewrite  5)
5)  # (x |-| x) = dbl (# x) ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

Again, straightforward when using xrewrite, but I am unable to complete the proof with just rewrite. The above proof shows the reason for this quite clearly: the prover does not realize that sequent 4 is trivially true by instantiating `x` in the antecedent with `x |- x'el`; therefore we need the antecedent instantiation capabilities of xrewrite to complete the last proof step.

4.4.7 $even(x + x)$

This and the following three theorems are easily proved in five steps each, using rewrite.

```
1)  ||-- forall(x:Nat) even (x + x) (induct  2,4)
2)  ||-- even (0 + 0) (rewrite  3)
3)  ||-- true (TRIV)
4)  even (x + x) ||-- even ((S x) + S x) (rewrite  5)
5)  even (x + x) ||-- even (x + x) (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.8 $odd(S(x) + x)$

```
1)  ||-- forall(x:Nat) odd ((S x) + x) (induct  2,4)
2)  ||-- odd (1 + 0) (rewrite  3)
3)  ||-- true (TRIV)
4)  odd ((S x) + x) ||-- odd ((S (S x)) + S x) (rewrite  5)
5)  odd ((S x) + x) ||-- odd ((S x) + x) (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.9 $even'(x + x)$

```
1)  ||-- forall(x:Nat) even' (x + x) (induct  2,4)
2)  ||-- even' (0 + 0) (rewrite  3)
3)  ||-- true (TRIV)
4)  even' (x + x) ||-- even' ((S x) + S x) (rewrite  5)
5)  even' (x + x) ||-- even' (x + x) (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.10 $odd'(S(x) + x)$

```
1)  ||-- forall(x:Nat) odd' ((S x) + x) (induct  2,4)
2)  ||-- odd' (1 + 0) (rewrite  3)
3)  ||-- true (TRIV)
4)  odd' ((S x) + x) ||-- odd' ((S (S x)) + S x) (rewrite  5)
5)  odd' ((S x) + x) ||-- odd' ((S x) + x) (TRIV)
The proof consists of 5 nodes.
```

```
Q.E.D.
```

4.4.11 $even'(x) -> half(x) + half(x) = x$

```
1)  ||-- forall(x:Nat) even' x => half x + half x = x
 (induct  2,4)
2)  ||-- even' 0 => half 0 + half 0 = 0 (rewrite  3)
3)  ||-- true (TRIV)
4)  even' x => half x + half x = x ||-- even' (S x) =>
 half (S x) + half (S x) = S x (xrewrite  5)
5)  even' x => half x + half x = x ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

This proof is equally simple to the preceding four, except I had to use xrewrite in the last rewriting step. Using only rewrite, I did not manage to prove the induction step, as shown below.

```
1)  ||-- forall(x:Nat) even' x => half x + half x = x
 (induct  2,4)
2)  ||-- even' 0 => half 0 + half 0 = 0 (rewrite  3)
3)  ||-- true (TRIV)
4)  even' x => half x + half x = x ||-- even' (S x) =>
 half (S x) + half (S x) = S x (rewrite  5)
5)  even' x => half x + half x = x ||-- odd' x =>
 half (S x) + half (S x) = S x (timpl  6)
6)  even' x => half x + half x = x,  odd' x
 ||-- half (S x) + half (S x) = S x (aimpl  7,9)
7)  ~ (even' x),  odd' x ||-- half (S x) + half (S x) = S x
 (anot  8)
8)  odd' x ||-- half (S x) + half (S x) = S x,  even' x
9)  half x + half x = x,  odd' x
 ||-- half (S x) + half (S x) = S x
```

4.4.12 $half(x + x) = x$

The two next proofs is similar to the above series of simple proofs, in that they do not require xrewrite.

```
1)  ||-- forall(x:Nat) half (x + x) = x (induct  2,4)
2)  ||-- half (0 + 0) = 0 (rewrite  3)
3)  ||-- true (TRIV)
4)  half (x + x) = x ||-- half ((S x) + S x) = S x (rewrite  5)
5)  half (x + x) = x ||-- half (x + x) = x (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.13 $half(S(x) + x) = x$

```
1)  ||-- forall(x:Nat) half ((S x) + x) = x (induct  2,4)
2)  ||-- half (1 + 0) = 0 (rewrite  3)
3)  ||-- true (TRIV)
4)  half ((S x) + x) = x ||-- half ((S (S x)) + S x) = S x
 (rewrite  5)
5)  half ((S x) + x) = x ||-- half ((S x) + x) = x (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.14 $rot(len(x), x) = x$

```
1)  ||-- forall(x:Seq{T}) rot (# x,x) = x (induct  2,4)
2)  ||-- rot (# e,e) = e (rewrite  3)
3)  ||-- true (TRIV)
4)  rot (# x,x) = x ||-- rot (# (x |- x'el),x |- x'el) =
 x |- x'el (xrewrite  5)
5)  rot (# x,x) = x ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

Here again, the ability of xrewrite to make rewrite rules from the antecedent is what makes the last rewrite step able to complete the proof; indeed, they all are provable using the command list (induct 1, rewrite, xrewrite).

The rest of the theorems are easily proved using xrewrite; most of them are proved with the command sequence (induct 1, rewrite, xrewrite), while some need another induction. That is, all can be proved using the command

repeat [xrewrite, induct +]

which (as described in section 2.2.6) repeatedly tries xrewrite, and then, if no rewriting is possible, induct in any (but only one) part of the consequent.

4.4.15 $len(rot(len(x), x)) = len(x)$

```
1)  ||-- forall(x:Seq{T}) # (rot (# x,x)) = # x (induct  2,4)
2)  ||-- # (rot (# e,e)) = # e (rewrite  3)
3)  ||-- true (TRIV)
4)  # (rot (# x,x)) = # x ||-- # (rot (# (x |- x'el),x |- x'el)) =
 # (x |- x'el) (xrewrite  5)
5)  # (rot (# x,x)) = # x ||-- true (TRIV)
```

```
The proof consists of 5 nodes.


Q.E.D.
```

4.4.16 $rot(len(x), x@[y]) = y :: rot(len(x), x)$

Here—and in the rest of the theorems incorporating sequence operations—all the sequence operators are mirrored, as described at the start of this chapter, to accommodate the fact that right concatenation is the sequence generator in ABEL, while left concatenation is in SPIKE.

```
1)  ||-- forall(x:Seq{T},y:T) rot (# x,e |- y |-| x) =
 rot (# x,x) |- y (induct  2,4)
2)  ||-- forall(y:T) rot (# e,e |- y |-| e) = rot (# e,e) |- y
 (rewrite  3)
3)  ||-- true (TRIV)
4)  forall(y:T) rot (# x,e |- y |-| x) = rot (# x,x) |- y
 ||-- forall(y:T) rot (# (x |- x'el),e |- y |-| (x |- x'el)) =
 rot (# (x |- x'el),x |- x'el) |- y (xrewrite  5)
5)  forall(y:T) rot (# x,e |- y |-| x) = rot (# x,x) |- y
 ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.17 $len(rev(x)) = len(x)$

```
1)  ||-- forall(x:Seq{T}) # (rev x) = # x (induct  2,4)
2)  ||-- # (rev e) = # e (rewrite  3)
3)  ||-- true (TRIV)
4)  # (rev x) = # x ||-- # (rev (x |- x'el)) = # (x |- x'el)
 (xrewrite  5)
5)  # (rev x) = # x ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.18 $rev(rev(x)) = x$

```
1)  ||-- forall(x:Seq{T}) rev (rev x) = x (induct  2,4)
2)  ||-- rev (rev e) = e (rewrite  3)
3)  ||-- true (TRIV)
4)  rev (rev x) = x ||-- rev (rev (x |- x'el)) = x |- x'el
 (xrewrite  5)
5)  rev (rev x) = x ||-- true (TRIV)
```

```
The proof consists of 5 nodes.


Q.E.D.
```

4.4.19 $rev(rev(x)@[y]) = y :: x$

```
1)  ||-- forall(x:Seq{T},y:T) rev ((e |- y) |-| rev x) = x |- y
 (induct  2,4)
2)  ||-- forall(y:T) rev ((e |- y) |-| rev e) = e |- y (rewrite  3)
3)  ||-- true (TRIV)
4)  forall(y:T) rev ((e |- y) |-| rev x) = x |- y
 ||-- forall(y:T) rev ((e |- y) |-| rev (x |- x'el)) = x |- x'el |- y
 (xrewrite  5)
5)  forall(y:T) rev ((e |- y) |-| rev x) = x |- y
 ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.20 $rev(rev(x)@[y]) = y :: rev(rev(x))$

```
1)  ||-- forall(x:Seq{T},y:T) rev ((e |- y) |-| rev x) =
 rev (rev x) |- y (induct  2,4)
2)  ||-- forall(y:T) rev ((e |- y) |-| rev e) =
 rev (rev e) |- y (rewrite  3)
3)  ||-- true (TRIV)
4)  forall(y:T) rev ((e |- y) |-| rev x) = rev (rev x) |- y
 ||-- forall(y:T) rev ((e |- y) |-| rev (x |- x'el)) =
 rev (rev (x |- x'el)) |- y (xrewrite  5)
5)  forall(y:T) rev ((e |- y) |-| rev x) = rev (rev x) |- y
 ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.21 $len(rev(x@y)) = len(x) + len(y)$

```
1)  ||-- forall(x:Seq{T},y:Seq{T}) # (rev (x |-| y)) =
 # x + # y (induct  2,8)
2)  ||-- forall(y:Seq{T}) # (rev (e |-| y)) = # e + # y
 (rewrite  3)
3)  ||-- forall(y:Seq{T}) # (rev (e |-| y)) = # y (induct  4,6)
4)  ||-- # (rev (e |-| e)) = # e (rewrite  5)
5)  ||-- true (TRIV)
6)  # (rev (e |-| y)) = # y ||-- # (rev (e |-| (y |- y'el))) =
 # (y |- y'el) (xrewrite  7)
```

```
7)  # (rev (e |-| y)) = # y ||-- true (TRIV)
8)  forall(y:Seq{T}) # (rev (x |-| y)) = # x + # y
 ||-- forall(y:Seq{T}) # (rev ((x |- x'el) |-| y)) =
 # (x |- x'el) + # y (xrewrite  9)
9)  forall(y:Seq{T}) # (rev (x |-| y)) = # x + # y
 ||-- true (TRIV)
The proof consists of 9 nodes.


Q.E.D.
```

4.4.22 $len(qrev(x, []))$ = $len(x)$

```
1)  ||-- forall(x:Seq{T}) # (qrev (x,e)) = # x (induct  2,4)
2)  ||-- # (qrev (e,e)) = # e (rewrite  3)
3)  ||-- true (TRIV)
4)  # (qrev (x,e)) = # x ||-- # (qrev (x |- x'el,e)) =
 # (x |- x'el) (xrewrite  5)
5)  # (qrev (x,e)) = # x ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.23 $qrev(x, y) = rev(x)@y$

```
1)  ||-- forall(x:Seq{T},y:Seq{T}) qrev (x,y) = y |-| rev x
 (induct  2,4)
2)  ||-- forall(y:Seq{T}) qrev (e,y) = y |-| rev e (rewrite  3)
3)  ||-- true (TRIV)
4)  forall(y:Seq{T}) qrev (x,y) = y |-| rev x
 ||-- forall(y:Seq{T}) qrev (x |- x'el,y) = y |-| rev (x |- x'el)
 (xrewrite  5)
5)  forall(y:Seq{T}) qrev (x,y) = y |-| rev x ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.24 $len(qrev(x, y)) = len(x) + len(y)$

```
1)  ||-- forall(x:Seq{T},y:Seq{T}) # (qrev (x,y)) = # x + # y
 (induct  2,4)
2)  ||-- forall(y:Seq{T}) # (qrev (e,y)) = # e + # y (rewrite  3)
3)  ||-- true (TRIV)
4)  forall(y:Seq{T}) # (qrev (x,y)) = # x + # y
 ||-- forall(y:Seq{T}) # (qrev (x |- x'el,y)) =
 # (x |- x'el) + # y (xrewrite  5)
5)  forall(y:Seq{T}) # (qrev (x,y)) = # x + # y
```

```
 ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.25 $qrev(qrev(x,[]),[]) = x$

```
1)  ||-- forall(x:Seq{T}) qrev (qrev (x,e),e) = x (induct  2,4)
2)  ||-- qrev (qrev (e,e),e) = e (rewrite  3)
3)  ||-- true (TRIV)
4)  qrev (qrev (x,e),e) = x ||-- qrev (qrev (x |- x'el,e),e) =
 x |- x'el (xrewrite  5)
5)  qrev (qrev (x,e),e) = x ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.26 $rev(qrev(x,[])) = x$

```
1)  ||-- forall(x:Seq{T}) rev (qrev (x,e)) = x (induct  2,4)
2)  ||-- rev (qrev (e,e)) = e (rewrite  3)
3)  ||-- true (TRIV)
4)  rev (qrev (x,e)) = x ||-- rev (qrev (x |- x'el,e)) =
 x |- x'el (xrewrite  5)
5)  rev (qrev (x,e)) = x ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.27 $qrev(rev(x),[]) = x$

```
1)  ||-- forall(x:Seq{T}) qrev (rev x,e) = x (induct  2,4)
2)  ||-- qrev (rev e,e) = e (rewrite  3)
3)  ||-- true (TRIV)
4)  qrev (rev x,e) = x ||-- qrev (rev (x |- x'el),e) =
 x |- x'el (xrewrite  5)
5)  qrev (rev x,e) = x ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.28 $nth(i,nth(j,x)) = nth(j,nth(i,x))$

```
1)  ||-- forall(i:Nat,j:Nat,x:Seq{T}) nth (i,nth (j,x)) =
 nth (j,nth (i,x)) (induct  2,4)
```

```
2)  ||-- forall(j:Nat,x:Seq{T}) nth (0,nth (j,x)) =
 nth (j,nth (0,x)) (rewrite  3)
3)  ||-- true (TRIV)
4)  forall(j:Nat,x:Seq{T}) nth (i,nth (j,x)) = nth (j,nth (i,x))
 ||-- forall(j:Nat,x:Seq{T}) nth (S i,nth (j,x)) =
 nth (j,nth (S i,x)) (xrewrite  5)
5)  forall(j:Nat,x:Seq{T}) nth (i,nth (j,x)) = nth (j,nth (i,x))
 ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.4.29 $nth(i, nth(j, nth(k, x))) = nth(k, nth(j, nth(i, x)))$

```
1)  ||-- forall(i:Nat,j:Nat,k:Nat,x:Seq{T})
 nth (i,nth (j,nth (k,x))) = nth (k,nth (j,nth (i,x)))
 (induct  2,8)
2)  ||-- forall(j:Nat,k:Nat,x:Seq{T}) nth (0,nth (j,nth (k,x))) =
 nth (k,nth (j,nth (0,x))) (rewrite  3)
3)  ||-- forall(j:Nat,k:Nat,x:Seq{T}) nth (j,nth (k,x)) =
 nth (k,nth (j,x)) (induct  4,6)
4)  ||-- forall(k:Nat,x:Seq{T}) nth (0,nth (k,x)) =
 nth (k,nth (0,x)) (xrewrite  5)
5)  ||-- true (TRIV)
6)  forall(k:Nat,x:Seq{T}) nth (j,nth (k,x)) = nth (k,nth (j,x))
 ||-- forall(k:Nat,x:Seq{T}) nth (S j,nth (k,x)) =
 nth (k,nth (S j,x)) (xrewrite  7)
7)  forall(k:Nat,x:Seq{T}) nth (j,nth (k,x)) =
 nth (k,nth (j,x)) ||-- true (TRIV)
8)  forall(j:Nat,k:Nat,x:Seq{T}) nth (i,nth (j,nth (k,x))) =
 nth (k,nth (j,nth (i,x))) ||-- forall(j:Nat,k:Nat,x:Seq{T})
 nth (S i,nth (j,nth (k,x))) = nth (k,nth (j,nth (S i,x)))
 (xrewrite  9)
9)  forall(j:Nat,k:Nat,x:Seq{T}) nth (i,nth (j,nth (k,x))) =
 nth (k,nth (j,nth (i,x))) ||-- true (TRIV)
The proof consists of 9 nodes.


Q.E.D.
```

4.4.30 $len(isort(x)) = len(x)$

```
1)  ||-- forall(x:Seq{Int}) # (isort x) = # x (induct  2,4)
2)  ||-- # (isort e) = # e (rewrite  3)
3)  ||-- true (TRIV)
4)  # (isort x) = # x ||-- # (isort (x |- x'el)) = # (x |- x'el)
 (xrewrite  5)
5)  # (isort x) = # x ||-- true (TRIV)
```

```
The proof consists of 5 nodes.


Q.E.D.
```

4.4.31 $sorted(isort(x))$

```
1)  ||-- forall(x:Seq{Int}) sorted (isort x) (induct  2,4)
2)  ||-- sorted (isort e) (rewrite  3)
3)  ||-- true (TRIV)
4)  sorted (isort x) ||-- sorted (isort (x |- x'el))
 (xrewrite  5)
5)  sorted (isort x) ||-- true (TRIV)
The proof consists of 5 nodes.


Q.E.D.
```

4.5 Conclusions from the Tests

As mentioned above there is some rather fundamental differences between SPIKE and the ABEL system, due to the fact that SPIKE is an automatic system, while the other is an interactive one. This means, amongst others, that the user of the ABEL system has somewhat equivalent tasks (although not the same) to the "divergence critic" described in [Wal96]; namely, to see when the system gets stuck, and help it along by using diverse techniques and strategies. It also means that while the ABEL system has to take care to keep the proof as simple and clean as possible to make it as easy as possible for the user to follow the reasoning, SPIKE and other automatic provers have the potential benefit of the possibility to reason in a way unfit for human reading. Notwithstanding, it is of course a goal that the proof system should do as much as possible without the need of user interference.

Nevertheless, we see that by using only simple commands all the theorems can be easily proved with the ABEL system. No real logical insight on the part of the user was needed in the previous examples.

Compared to the results in [Wal96], the ABEL system fared as well as could be hoped; the command sequences needed to complete the proofs are quite simple, and as we shall see later, can to a relatively high degree be automated.

Chapter 5

Some Further Proof Examples

In this chapter I will present a few examples of what I hope is realistic verification-related proofs. Some are from exams in the course IN 217 (Program Specification and Verification) at the Department of Informatics, University of Oslo[1]; some from the compendium to IN 217. One of the design premises for the ABEL system was that the system was to be a tool for use in this course.

All of these proofs are constructed in insignificant time by the system; the execution time is far dominated by the time it takes the user to operate the system.

5.1 A Simple Switching Loop

I found this example in the solution for assignment 3-b in the 1993 exam for IN 217. It is the proof of invariance for the following loop:

$$\textbf{for } k := 2 \textbf{ to } n \textbf{ do } a[1] :=: a[k] \textbf{ od}$$

Here, the :=: operator is parallel assignment (swapping)—after a statement of "x :=: y", x has y's previous value, and y has x's. The rest of the syntax should be trivial.

We have as invariant I:

$$a = a0[k-1] \dashv a0[1..k-2] \vdash a0[k..n]$$

and the Hoare logic rule SFOR gives us the premise:

$$\{s \leq k \leq n \wedge I\}\ a[1] :=: a[k]\ \{I^k_{k+1}\}$$

[1]The assignments can be found on the web at
`http://www.ifi.uio.no/studinf/eksamen/eksoppg/in217/`

This gives us the following verification obligation:

$$\forall(a : \text{Seq\{T\}}, a0 : \text{Seq\{T\}}, k : \text{Int}, n : \text{Int})\ 2 \leq k \leq n \wedge$$
$$a = a0[k-1] \dashv a0[1..k-2] \vdash a0[k..n] \Rightarrow$$
$$a[1 \rightarrow a[k]][k \rightarrow a[1]] = a0[k] \dashv a0[1..k-1] \vdash a0[k+1..n]$$

This seems to me to be a somewhat realistic example of verification lemmas, albeit a bit simple. As the following proof shows, this lemma was proved in nine steps with the command repeat [xrewrite, induct +].

```
1)  ||-- forall(a:Seq{T},a0:Seq{T},k:Int,n:Int) 2 <= k /\ k
 <= n /\ a = a0[k - 1] -| a0[1 .. k - 2] |-| a0[k .. n] =>
 a[1 -> a[k]][k -> a[1]] = a0[k] -| a0[1 .. k - 1] |-|
 a0[k + 1 .. n] (xrewrite  2)
2)  ||-- forall(a:Seq{T},a0:Seq{T},k:Int,n:Int)
 (2 < k \/ 2 = k) /\ (k < n \/ k = n) /\ a = a0[k - 1] -|
 a0[1 .. k - 2] |-| a0[k .. n] => a[1 -> a[k]][k -> a[1]] =
 a0[k] -| a0[1 .. k - 1] |-| a0[k + 1 .. n] (induct  3,8)
3)  ||-- forall(a0:Seq{T},k:Int,n:Int) (2 < k \/ 2 = k) /\
 (k < n \/ k = n) /\ e = a0[k - 1] -| a0[1 .. k - 2] |-|
 a0[k .. n] => e[1 -> e[k]][k -> e[1]] =
 a0[k] -| a0[1 .. k - 1] |-| a0[k + 1 .. n] (induct  4,6)
4)  ||-- forall(k:Int,n:Int) (2 < k \/ 2 = k) /\
 (k < n \/ k = n) /\ e = e[k - 1] -| e[1 .. k - 2] |-|
 e[k .. n] => e[1 -> e[k]][k -> e[1]] = e[k] -|
 e[1 .. k - 1] |-| e[k + 1 .. n] (xrewrite  5)
5)  ||-- true (TRIV)
6)  forall(k:Int,n:Int) (2 < k \/ 2 = k) /\
 (k < n \/ k = n) /\ e = a0[k - 1] -| a0[1 .. k - 2] |-|
 a0[k .. n] => e[1 -> e[k]][k -> e[1]] = a0[k] -|
 a0[1 .. k - 1] |-| a0[k + 1 .. n] ||-- forall(k:Int,n:Int)
 (2 < k \/ 2 = k) /\ (k < n \/ k = n) /\ e =
 (a0 |- a0'el)[k - 1] -| (a0 |- a0'el)[1 .. k - 2] |-|
 (a0 |- a0'el)[k .. n] => e[1 -> e[k]][k -> e[1]] =
 (a0 |- a0'el)[k] -| (a0 |- a0'el)[1 .. k - 1] |-|
 (a0 |- a0'el)[k + 1 .. n] (xrewrite  7)
7)  forall(k:Int,n:Int) (2 < k \/ 2 = k) /\ (k < n \/ k = n) /\
 e = a0[k - 1] -| a0[1 .. k - 2] |-| a0[k .. n] =>
 e[1 -> e[k]][k -> e[1]] = a0[k] -| a0[1 .. k - 1] |-|
 a0[k + 1 .. n] ||-- true (TRIV)
8)  forall(a0:Seq{T},k:Int,n:Int) (2 < k \/ 2 = k) /\
 (k < n \/ k = n) /\ a = a0[k - 1] -| a0[1 .. k - 2] |-|
 a0[k .. n] => a[1 -> a[k]][k -> a[1]] = a0[k] -|
 a0[1 .. k - 1] |-| a0[k + 1 .. n]
 ||-- forall(a0:Seq{T},k:Int,n:Int) (2 < k \/ 2 = k) /\
 (k < n \/ k = n) /\ a |- a'el = a0[k - 1] -|
 a0[1 .. k - 2] |-| a0[k .. n] =>
 (a |- a'el)[1 -> (a |- a'el)[k]][k -> (a |- a'el)[1]] =
```

```
 a0[k] -| a0[1 .. k - 1] |-| a0[k + 1 .. n] (xrewrite  9)
9)  forall(a0:Seq{T},k:Int,n:Int) (2 < k \/ 2 = k) /\
 (k < n \/ k = n) /\ a = a0[k - 1] -| a0[1 .. k - 2] |-|
 a0[k .. n] => a[1 -> a[k]][k -> a[1]] = a0[k] -|
 a0[1 .. k - 1] |-| a0[k + 1 .. n] ||-- true (TRIV)
The proof consists of 9 nodes.


Q.E.D.
```

The sequent grows rather large, partly because the ABEL system rewrites $x \leq y$ into $(x < y) \vee (x = y)$. As we see, two inductions was needed to complete this proof. However, if one inducts over k (currently one has to put the variable textually first in the universal quantifier to do this), the proof is completed in eight steps with just one induction. This exemplifies the need for a facility for specifying which variable to induct over.

5.2 Transitivity of the Sub-Tree Relation

This proof proves transitivity of the _sub_ function from figure 3.1. The first proof shows induction over the *Tree* type's three generators, while the second shows that using BPC from the start here generates a smaller proof.

The lemma to be proved:

$$\forall(s, r, t : \text{Tree}\{T\})\ s \text{ \_sub\_ } r \wedge r \text{ \_sub\_ } t \Rightarrow s \text{ \_sub\_ } t$$

First, starting with induction:

```
1)  ||-- forall(s:Tree{T},r:Tree{T},t:Tree{T})
 s _sub_ r /\ r _sub_ t => s _sub_ t (induct  2,8,14)
2)  ||-- forall(r:Tree{T},t:Tree{T}) nil _sub_ r /\
 r _sub_ t => nil _sub_ t (tall  3)
3)  ||-- forall(t:Tree{T}) nil _sub_ r /\ r _sub_ t =>
 nil _sub_ t (tall  4)
4)  ||-- nil _sub_ r /\ r _sub_ t => nil _sub_ t (timpl  5)
5)  nil _sub_ r /\ r _sub_ t ||-- nil _sub_ t (aand  6)
6)  nil _sub_ r,  r _sub_ t ||-- nil _sub_ t (xrewrite  7)
7)  nil _sub_ r,  r _sub_ t ||-- true (TRIV)
8)  ||-- forall(r:Tree{T},t:Tree{T}) leaf s'el _sub_ r /\
 r _sub_ t => leaf s'el _sub_ t (tall  9)
9)  ||-- forall(t:Tree{T}) leaf s'el _sub_ r /\
 r _sub_ t => leaf s'el _sub_ t (tall  10)
10)  ||-- leaf s'el _sub_ r /\ r _sub_ t =>
 leaf s'el _sub_ t (timpl  11)
11)  leaf s'el _sub_ r /\ r _sub_ t ||--
 leaf s'el _sub_ t (aand  12)
```

```
12)  leaf s'el _sub_ r,  r _sub_ t ||--
 leaf s'el _sub_ t (xrewrite  13)
13)  leaf s'el _sub_ r,  r _sub_ t ||-- true (TRIV)
14)  forall(r:Tree{T},t:Tree{T}) s _sub_ r /\ r _sub_ t =>
 s _sub_ t ||-- forall(r:Tree{T},t:Tree{T}) tree (s,s) _sub_ r /\
 r _sub_ t => tree (s,s) _sub_ t (xrewrite  15)
15)  forall(r:Tree{T},t:Tree{T}) s _sub_ r /\
 r _sub_ t => s _sub_ t ||-- true (TRIV)
The proof consists of 15 nodes.


Q.E.D.
```

This proof went relatively easy, but what happens if we try starting with bpc?

```
1)  ||-- forall(s:Tree{T},r:Tree{T},t:Tree{T}) s _sub_ r /\
 r _sub_ t => s _sub_ t (tall  2)
2)  ||-- forall(r:Tree{T},t:Tree{T}) s _sub_ r /\
 r _sub_ t => s _sub_ t (tall  3)
3)  ||-- forall(t:Tree{T}) s _sub_ r /\ r _sub_ t =>
 s _sub_ t (tall  4)
4)  ||-- s _sub_ r /\ r _sub_ t => s _sub_ t (timpl  5)
5)  s _sub_ r /\ r _sub_ t ||-- s _sub_ t (aand  6)
6)  s _sub_ r,  r _sub_ t ||-- s _sub_ t (xrewrite  7)
7)  s _sub_ r,  r _sub_ t ||-- true (TRIV)
The proof consists of 7 nodes.


Q.E.D.
```

Starting with bpc the proof is less than half the length compared to starting with induction—while normally, using induction leads to a shorter proof. Here, the difference in length is due to that xrewrite is able to prove the transitivity in one step once bpc has "massaged" the sequent. However, I cannot see a simple way for the user to see before starting which proof strategy would yield the shortest proof here—and anyway, deciding this would probably take longer time than just trying it out.

5.3 Sequential Search for a Given Value

The following four subproofs are to be found in Example 29, p. 69 in [Lin99] (Introduction to Hoare logic and abstract types), the compendium used to supplement [Dah92] as curriculum for the previously mentioned course IN 217. The proofs are taken from the verification of the following loop for sequential search (for a given x) through an array, here adorned with Hoare sentences:

s := 0;
$\{s = 0\}$
L: **for** k := 1 **to** n **do**
{PRE}
if $A[k]$ = x **then**
s := k;
exit L
fi
{POST}
od
{**if** $1 \leq s \leq n$ **then** $A[s] = x$ **else** $\forall(i : \text{Nat})\ i \leq n \Rightarrow A[i] \neq x$ **fi**}

The example proposes the following pre- and postinvariants for the loop, as shown above:

Preinvariant:

PRE: **if** $1 \leq s \leq n$ **then** $A[s] = x$ **else** $\forall(i : \text{Nat})\ i < k \Rightarrow A[i] \neq x$ **fi**

Post invariant:

POST: **if** $1 \leq s \leq n$ **then** $A[s] = x$ **else** $\forall(i : \text{Nat})\ i \leq k \Rightarrow A[i] \neq x$ **fi**

Proof Burdens The FOR rule (from Hoare logic) applied to the code above results in three proof obligations:

1. $\{1 \leq k \leq n \wedge \text{PRE}\}$ **if** ... **fi**$\{POST\}$
2. $1 \leq k < n \wedge \text{POST} \Rightarrow \text{PRE}^k_{k+1}$
3. $1 > n \wedge \text{PRE}^k_1 \Rightarrow \text{POST}^k_n$

5.3.1 Proof obligation 1

The Hoare logic rule TDSHIF (top down short if) applied to proof obligation 1 above produces two new obligations,

1.1. $\{1 \leq k \leq n \wedge \text{PRE} \wedge A[k] = x\} s := k; \textbf{exit} L \{\text{POST}\}$

1.2. $1 \leq k \leq n \wedge \text{PRE} \wedge A[k] \neq x \Rightarrow \text{POST}$

By the Hoare rule SEQ (sequence of statements) we may further transform obligation 1.1 into the two obligations

1.1.1. $\{1 \leq k \leq n \wedge \text{PRE} \wedge A[k] = x\}\ s := k\ \{A[s] = x \wedge 1 \leq s \leq n\}$

1.1.2. $\{A[s] = x \wedge 1 \leq s \leq n\}$ **exit** L {POST}

Obligation 1.1.2 is trivially proved, so I will not include its proof here. Obligation 1.1.1 is transformed by use of AS (assignment) and CQL (left consequence) into

$$\forall(k, n, s, x : \text{Nat}, A : \text{Seq\{Nat\}})\ 1 \leq k \leq n \wedge$$
$$\textbf{if } 1 \leq s \leq n \textbf{ then } A[s] = x \textbf{ else } \forall(i : \text{Nat})\ i < k \Rightarrow A[i] \neq x \textbf{ fi} \wedge$$
$$A[k] = x \Rightarrow$$
$$A[k] = x \wedge 1 \leq k \leq n$$

Written out in ABEL, obligation 1.2 becomes

$$\forall(k, n, s, x : \text{Nat}, A : \text{Seq\{Nat\}})\ 1 \leq k \leq n \wedge$$
$$\textbf{if } 1 \leq s \leq n \textbf{ then } A[s] = x \textbf{ else } \forall(i : \text{Nat})\ i < k \Rightarrow A[i] \neq x \textbf{ fi} \wedge$$
$$A[k] \neq x \Rightarrow \textbf{if } 1 \leq s \leq n \textbf{ then } A[s] = x \textbf{ else } \forall(i : \text{Nat})\ i \leq k \Rightarrow A[i] \neq x \textbf{ fi}$$

Obligation 1.1.1

This proof obligation can be successfully proved in the ABEL system both inductively and using BPC reasoning. More succinctly, both the command repeat [xrewrite, induct +] and repeat [xrewrite, bpc] leads to success. As can be witnessed from the following, using induction leads to significantly fewer steps compared to BPC; however, using BPC takes less execution time.

Induction:

```
1)  ||-- forall(k:Nat,n:Nat,s:Nat,x:Nat,A:Seq{Nat})
 1 <= k /\ k <= n /\
 if 1 <= s /\ s <= n
   then A[s] = x
   else forall(i:Nat) i < k => A[i] =/= x
 fi /\ A[k] = x => A[k] = x /\ 1 <= k /\ k <= n (xrewrite  2)
2)  ||-- forall(k:Nat,n:Nat,s:Nat,x:Nat,A:Seq{Nat})
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi /\ A[k] = x => A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (induct  3,5)
3)  ||-- forall(n:Nat,s:Nat,x:Nat,A:Seq{Nat})
 (1 < 0 \/ 1 = 0) /\ (0 < n \/ 0 = n) /\
```

```
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 0 =>
        if A[i] = x then false else true fi
 fi /\ A[0] = x => A[0] = x /\ (1 < 0 \/ 1 = 0) /\
 (0 < n \/ 0 = n) (xrewrite  4)
4)  ||-- true (TRIV)
5)  forall(n:Nat,s:Nat,x:Nat,A:Seq{Nat})
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi /\ A[k] = x => A[k] = x /\ (1 < k \/ 1 = k)
 /\ (k < n \/ k = n) ||--
 forall(n:Nat,s:Nat,x:Nat,A:Seq{Nat})
 (1 < S k \/ 1 = S k) /\ (S k < n \/ S
   k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < S k =>
        if A[i] = x then false else true fi
 fi /\ A[S k] = x => A[S k] = x /\ (1 < S k \/ 1 = S k) /\
 (S k < n \/ S k = n) (xrewrite  6)
6)  forall(n:Nat,s:Nat,x:Nat,A:Seq{Nat})
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi /\ A[k] = x => A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) ||-- true (TRIV)
The proof consists of 6 nodes.


Q.E.D.
```

The proof used 0.11 seconds when executed on a Sun Ultra 1 with 184 MB RAM and 167 MHz UltraSPARC CPU, running Solaris 7. This computer was also used for all the subsequent timings.

BPC:

```
1)  ||-- forall(k:Nat,n:Nat,s:Nat,x:Nat,A:Seq{Nat})
 1 <= k /\ k <= n /\
 if 1 <= s /\ s <= n
   then A[s] = x
   else forall(i:Nat) i < k => A[i] =/= x
 fi /\ A[k] = x => A[k] = x /\ 1 <= k /\ k <= n (xrewrite  2)
```

```
2)  ||-- forall(k:Nat,n:Nat,s:Nat,x:Nat,A:Seq{Nat})
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi /\ A[k] = x => A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (tall  3)
3)  ||-- forall(n:Nat,s:Nat,x:Nat,A:Seq{Nat})
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi /\ A[k] = x => A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (tall  4)
4)  ||-- forall(s:Nat,x:Nat,A:Seq{Nat})
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi /\ A[k] = x => A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (tall  5)
5)  ||-- forall(x:Nat,A:Seq{Nat})
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi /\ A[k] = x => A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (tall  6)
6)  ||-- forall(A:Seq{Nat})
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi /\ A[k] = x => A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (tall  7)
7)  ||-- (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi /\ A[k] = x => A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (timpl  8)
8)  (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
```

```
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi /\ A[k] = x ||-- A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (aand  9)
9)  (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi,
 A[k] = x ||-- A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (aand  10)
10)  (1 < k \/ 1 = k) /\ (k < n \/ k = n),
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi,
 A[k] = x ||-- A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (aand  11)
11)  1 < k \/ 1 = k,
 k < n \/ k = n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi,
 A[k] = x ||-- A[k] = x /\ (1 < k \/ 1 = k) /\
 (k < n \/ k = n) (tand  12,15)
12)  1 < k \/ 1 = k,
 k < n \/ k = n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi,
 A[k] = x ||-- A[k] = x /\ (1 < k \/ 1 = k) (tand  13,14)
13)  1 < k \/ 1 = k,
 k < n \/ k = n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi,
 A[k] = x ||-- A[k] = x (TRIV)
14)  1 < k \/ 1 = k,
 k < n \/ k = n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
```

```
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi,
 A[k] = x ||-- 1 < k \/ 1 = k (TRIV)
15)  1 < k \/ 1 = k,
 k < n \/ k = n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
        if A[i] = x then false else true fi
 fi,
 A[k] = x ||-- k < n \/ k = n (TRIV)
The proof consists of 15 nodes.


Q.E.D.
```

This proof took 0.05 seconds, less than half of the inductive proof. It was done with the command sequence (xrewrite, bpc), while the inductive proof was done with (xrewrite, induct 1, xrewrite, xrewrite) (one rewrite per induction step). That is, the inductive proof is shorter in the number of steps, while the deductive proof takes less time.

Obligation 1.2

For clarity, here is the obligation repeated:

$$\forall(k, n, s, x : \text{Nat}, A : \text{Seq}\{\text{Nat}\})\ 1 \leq k \leq n \wedge$$
$$\textbf{if } 1 \leq s \leq n \textbf{ then } A[s] = x \textbf{ else } \forall(i : \text{Nat})\ i < k \Rightarrow A[i] \neq x \textbf{ fi} \wedge$$
$$A[k] \neq x \Rightarrow \textbf{if } 1 \leq s \leq n \textbf{ then } A[s] = x \textbf{ else } \forall(i : \text{Nat})\ i \leq k \Rightarrow A[i] \neq x \textbf{ fi}$$

This is a typical loop verification theorem of a somewhat inductive nature; we want to prove that given some conditions (here, $1 \leq k$ and $k \leq n$), an expression including the relation $i < k$ implies the same expression with $i \leq k$. That is, given that the loop test succeeds and another iteration of the loop is executed, the loop invariant holds with a counter (here i) increased by one.

```
1)  ||-- forall(k:Nat,n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 1 <= k /\ k <= n /\
 if 1 <= s /\ s <= n
   then A[s] = x
   else forall(i:Nat) i < k => A[i] =/= x
 fi /\ A[k] =/= x =>
 if 1 <= s /\ s <= n
```

```
   then A[s] = x
   else forall(i:Nat)
   i <= k => A[i] =/= x fi (xrewrite  2)
2)  ||-- forall(k:Nat,n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
     if A[i] = x then false else true fi
 fi /\ if A[k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi (induct  3,5)
3)  ||-- forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < 0 \/ 1 = 0) /\ (0 < n \/ 0 = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 0 =>
     if A[i] = x then false else true fi
 fi /\ if A[0] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 0 \/ i = 0 =>
     if A[i] = x then false else true fi
 fi (xrewrite  4)
4)  ||-- true (TRIV)
5)  forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
     if A[i] = x then false else true fi
 fi /\ if A[k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < S k \/ 1 = S k) /\ (S k < n \/ S k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < S k =>
     if A[i] = x then false else true fi
 fi /\ if A[S k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < S k \/ i = S k =>
```

```
        if A[i] = x then false else true fi
 fi (induct  6,8)
6)  forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
     if A[i] = x then false else true fi
 fi /\ if A[k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- forall(s:Nat,A:Seq{Nat},x:Nat)
 (1 < S k \/ 1 = S k) /\ (S k < 0 \/ S k =
 0) /\
 if (1 < s \/ 1 = s) /\ (s < 0 \/ s = 0)
   then A[s] = x
   else forall(i:Nat) i < S k =>
     if A[i] = x then false else true fi
 fi /\ if A[S k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < 0 \/ s = 0)
   then A[s] = x
   else forall(i:Nat) i < S k \/ i = S k =>
        if A[i] = x then false else true fi
 fi (xrewrite  7)
7)  forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
     if A[i] = x then false else true fi
 fi /\ if A[k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- true (TRIV)
8)  forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
     if A[i] = x then false else true fi
 fi /\ if A[k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
```

```
 fi,
 forall(s:Nat,A:Seq{Nat},x:Nat)
 (1 < S k \/ 1 = S k) /\ (S k < n \/ S k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < S k =>
     if A[i] = x then false else true fi
 fi /\ if A[S k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < S k \/ i = S k =>
        if A[i] = x then false else true fi
 fi ||-- forall(s:Nat,A:Seq{Nat},x:Nat)
 (1 < S k \/ 1 = S k) /\ (S k < S n \/ S k = S n) /\
 if (1 < s \/ 1 = s) /\ (s < S n \/ s = S n)
   then A[s] = x
   else forall(i:Nat) i < S k =>
     if A[i] = x then false else true fi
 fi /\ if A[S k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < S n \/ s = S n)
   then A[s] = x
   else forall(i:Nat) i < S k \/ i = S k =>
        if A[i] = x then false else true fi
 fi (xrewrite  9)
9)  forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ (k < n \/ k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k =>
     if A[i] = x then false else true fi
 fi /\ if A[k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 forall(s:Nat,A:Seq{Nat},x:Nat)
 (0 < k \/ 0 = k) /\ (S k < n \/ S k = n) /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < S k =>
     if A[i] = x then false else true fi
 fi /\ if A[S k] = x then false else true fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < S k \/ i = S k =>
        if A[i] = x then false else true fi
 fi ||-- true (TRIV)
The proof consists of 9 nodes.
```

```
Q.E.D.
```

Execution time for this proof was 0.40 seconds on the same Ultra 1 as above. This proof seems long because of the sheer amount of text, but is only nine steps—the lenght is derived from the way the rewriter inflates expressions by e.g. rewriting $x \leq y$ to $x < y \vee x = y$, and the assumptions introduced by the two inductions.

This proof can also be performed deductively with bpc; this proof runs rather long at 211 steps, so I will not include it. With BPC deduction this proof took 1.24 seconds. As this proof may seem superficially similar to the last, it seems hard for the user to make quick guesses on whether induction or deduction will prove fastest for a certain theorem.

One issue to note with regard to proving this theorem with bpc is that the simple command repeat [xrewrite, bpc] does not work—at one point (specifically, branch 1.2), applying xrewrite leads to a sequent I have not been able to make the system prove.

5.3.2 Proof obligation 2

This obligation is the proof of loop invariance when the loop criterion is true (i.e. the loop body has been executed at least once, and will be executed again). It can be proved both using inductive and BPC reasoning, i.e. both using repeat [xrewrite, induct +] and repeat [xrewrite, bpc].

The theorem:

$$\forall(k, n, s, x : \text{Nat}, A : \text{Seq\{Nat\}})\ 1 \leq k < n \wedge$$
$$\textbf{if } 1 \leq s \leq n \textbf{ then } A[s] = x \textbf{ else } (\forall(i : \text{Nat})\ i \leq k \Rightarrow A[i] \neq x) \textbf{ fi} \Rightarrow$$
$$\textbf{if } 1 \leq s \leq n \textbf{ then } A[s] = x \textbf{ else } (\forall(i : \text{Nat})\ i < k + 1 \Rightarrow A[i] \neq x) \textbf{ fi}$$

As $i \leq k$ is equivalent to $i < k + 1$, the implication is quite trivial. However, the ABEL system has do "dig down" to these terms before being able to prove that, hence BPC proof is (as described below) quite long.

```
1)  ||-- forall(k:Nat,n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 1 <= k /\ k < n /\
 if 1 <= s /\ s <= n
   then A[s] = x
   else forall(i:Nat) i <= k => A[i] =/= x
 fi =>
 if 1 <= s /\ s <= n
   then A[s] = x
   else forall(i:Nat) i < k + 1 => A[i] =/= x
 fi (xrewrite  2)
```

```
2)  ||-- forall(k:Nat,n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (induct  3,5)
3)  ||-- forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < 0 \/ 1 = 0) /\ 0 < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 0 \/ i = 0 =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 0 + 1 =>
     if A[i] = x then false else true fi
 fi (xrewrite  4)
4)  ||-- true (TRIV)
5)  forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi ||-- forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < S k \/ 1 = S k) /\ S k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < S k \/ i = S k =>
        if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < S k + 1 =>
     if A[i] = x then false else true fi
 fi (xrewrite  6)
6)  forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
```

```
 (1 < k \/ 1 = k) /\ k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi ||-- true (TRIV)
The proof consists of 6 nodes.


Q.E.D.
```

In this proof induction is the faster technique—the system proves the obligation inductively in six steps, using just one induction. The proof generation had an execution time of 0.37 seconds, while the system uses 176 steps and 0.92 seconds for BPC. That is, BPC is here about 2.5 times slower than induction. I have included the BPC proof in appendix B.2.

5.3.3 Proof obligation 3

This proof shows a somewhat lengthy sequence of BPC rule applications. The proof obligation is:

$$\forall(k, n, s : \text{Nat}, A : \text{Seq\{T\}}, x : \text{Nat})\ 1 > n \wedge$$
$$\textbf{if } 1 \leq s \leq n \textbf{ then } A[s] = x \textbf{ else } \forall(i : \text{Nat})\ i < 1 \Rightarrow A[i] \neq x \textbf{ fi} \Rightarrow$$
$$\textbf{if } 1 \leq s \leq n \textbf{ then } A[s] = x \textbf{ else } \forall(i : \text{Nat})\ i \leq n \Rightarrow A[i] \neq x \textbf{ fi}$$

Again, this theorem is quite trivial if the quantifiers are dealt with. Handling the complexity of the theorem seems to be what makes the proof so long; a significant number of BPC rules has to be applied to do away with this complexity. This is reflected in that the theorem was proved with the simple command combination repeat [xrewrite, bpc]. An inductive proof was somewhat harder to find; by using two BPC rules the system completed the proof with the following command sequence (this proof is not included):

> xrewrite, induct 1, xrewrite, induct 1, xrewrite, induct 1, induct 1, timpl 1, induct 1, xrewrite, aall -1 ‘0‘, xrewrite, xrewrite, xrewrite, xrewrite, xrewrite, xrewrite

Here follows the deductive proof:

```
1)  ||-- forall(k:Nat,n:Nat,s:Nat,A:Seq{Nat},x:Nat) 1 > n /\
 if 1 <= s /\ s <= n
   then A[s] = x
   else forall(i:Nat) i < 1 => A[i] =/= x
 fi =>
 if 1 <= s /\ s <= n then A[s] = x else
   forall(i:Nat) i <= n => A[i] =/= x fi (xrewrite  2)
2)  ||-- forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat) n < 1 /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < n \/ i = n =>
     if A[i] = x then false else true fi
 fi (tall  3)
3)  ||-- forall(s:Nat,A:Seq{Nat},x:Nat) n < 1 /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < n \/ i = n =>
     if A[i] = x then false else true fi
 fi (tall  4)
4)  ||-- forall(A:Seq{Nat},x:Nat) n < 1 /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < n \/ i = n =>
     if A[i] = x then false else true fi
 fi (tall  5)
5)  ||-- forall(x:Nat) n < 1 /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < n \/ i = n =>
```

```
    if A[i] = x then false else true fi
 fi (tall  6)
6)  ||-- n < 1 /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < n \/ i = n =>
     if A[i] = x then false else true fi
 fi (timpl  7)
7)  n < 1 /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi ||-- if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < n \/ i = n =>
     if A[i] = x then false else true fi
 fi (aand  8)
8)  n < 1,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi ||-- if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < n \/ i = n =>
     if A[i] = x then false else true fi
 fi (tif  9,62)
9)  n < 1,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi ||-- (1 < s \/ 1 = s) /\ (s < n \/ s = n) =>
 A[s] = x (timpl  10)
10)  n < 1,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n)
 ||-- A[s] = x (aand  11)
11)  n < 1,
```

```
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi,
 1 < s \/ 1 = s,
 s < n \/ s = n ||-- A[s] = x (aor  12,37)
12)  n < 1,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi,
 1 < s,
 s < n \/ s = n ||-- A[s] = x (aor  13,25)
13)  n < 1,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi,
 1 < s,
 s < n ||-- A[s] = x (aif  14)
14)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 =>
  if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x (aimpl  15,24)
15)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x (anot  16)
16)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tand  17,19)
17)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  1 < s \/ 1 = s (tor  18)
18)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
```

```
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
19)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  s < n \/ s = n (aimpl  20,22)
20)  n < 1,  ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,  s < n ||-- A[s] = x,  s < n \/ s = n (tor  21)
21)  n < 1,  ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,  s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
22)  n < 1,
 forall(i:Nat) i < 1 => if A[i] = x then false else true fi,
 1 < s,
 s < n ||-- A[s] = x,  s < n \/ s = n (tor  23)
23)  n < 1,
 forall(i:Nat) i < 1 => if A[i] = x then false else true fi,
 1 < s,
 s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
24)  n < 1,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x (TRIV)
25)  n < 1,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi,
 1 < s,
 s = n ||-- A[s] = x (aif  26)
26)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x (aimpl  27,36)
27)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x (anot  28)
28)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
```

```
 1 < s, s = n ||-- A[s] = x,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tand  29,31)
29)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  1 < s \/ 1 = s (tor  30)
30)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
31)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  s < n \/ s = n (aimpl  32,34)
32)  n < 1,  ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,  s = n ||-- A[s] = x,  s < n \/ s = n (tor  33)
33)  n < 1,  ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,  s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
34)  n < 1,
 forall(i:Nat) i < 1 => if A[i] = x then false else true fi,
 1 < s,
 s = n ||-- A[s] = x,  s < n \/ s = n (tor  35)
35)  n < 1,
 forall(i:Nat) i < 1 => if A[i] = x then false else true fi,
 1 < s,
 s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
36)  n < 1,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x (TRIV)
37)  n < 1,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi,
 1 = s,
 s < n \/ s = n ||-- A[s] = x (aor  38,50)
38)  n < 1,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi,
```

```
 1 = s,
 s < n ||-- A[s] = x (aif  39)
39)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x (aimpl  40,49)
40)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x (anot  41)
41)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s, s < n ||-- A[s] = x,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tand  42,44)
42)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x,  1 < s \/ 1 = s (tor  43)
43)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
44)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x,  s < n \/ s = n (aimpl  45,47)
45)  n < 1,  ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,  s < n ||-- A[s] = x,  s < n \/ s = n (tor  46)
46)  n < 1,  ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,  s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
47)  n < 1,
 forall(i:Nat) i < 1 => if A[i] = x then false else true fi,
 1 = s,
 s < n ||-- A[s] = x,  s < n \/ s = n (tor  48)
48)  n < 1,
 forall(i:Nat) i < 1 => if A[i] = x then false else true fi,
 1 = s,
 s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
49)  n < 1,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
```

```
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x (TRIV)
50)  n < 1,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi,
 1 = s,
 s = n ||-- A[s] = x (aif  51)
51)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x (aimpl  52,61)
52)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x (anot  53)
53)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s, s = n ||-- A[s] = x,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tand  54,56)
54)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  1 < s \/ 1 = s (tor  55)
55)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
56)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  s < n \/ s = n (aimpl  57,59)
57)  n < 1,  ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,  s = n ||-- A[s] = x,  s < n \/ s = n (tor  58)
58)  n < 1,  ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,  s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
59)  n < 1,
 forall(i:Nat) i < 1 => if A[i] = x then false else true fi,
```

```
 1 = s,
 s = n ||-- A[s] = x,  s < n \/ s = n (tor  60)
60)  n < 1,
 forall(i:Nat) i < 1 => if A[i] = x then false else true fi,
 1 = s,
 s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
61)  n < 1,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x (TRIV)
62)  n < 1,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < 1 =>
     if A[i] = x then false else true fi
 fi ||-- ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < n \/ i = n =>
  if A[i] = x then false else true fi) (aif  63)
63)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi)
 ||-- ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < n \/ i = n =>
  if A[i] = x then false else true fi) (timpl  64)
64)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))
 ||-- forall(i:Nat) i < n \/ i = n =>
 if A[i] = x then false else true fi (anot  65)
65)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi)
 ||-- forall(i:Nat) i < n \/ i = n =>
 if A[i] = x then false else true fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tall  66)
66)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi)
 ||-- i < n \/ i = n => if A[i] = x then false else true fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (timpl  67)
67)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
```

```
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n \/ i = n ||-- if A[i] = x then false else true fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tand  68,92)
68)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n \/ i = n ||-- if A[i] = x then false else true fi,
 1 < s \/ 1 = s (tor  69)
69)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n \/ i = n ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aor  70,90)
70)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aimpl  71,88)
71)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (anot  72)
72)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n ||-- if A[i] = x then false else true fi,
 1 < s,
 1 = s,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tand  73,86)
73)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s,  1 < s \/ 1 = s (tor  74)
74)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aimpl  75,80)
75)  n < 1,  ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 i < n ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (anot  76)
76)  n < 1,  (1 < s \/ 1 = s) /\ (s < n \/ s = n),  i < n
```

```
 ||-- if A[i] = x then false else true fi,  1 < s,
 1 = s (aand  77)
77)  n < 1,  1 < s \/ 1 = s,  s < n \/ s = n,  i < n
 ||-- if A[i] = x then false else true fi,  1 < s,
 1 = s (aor  78,79)
78)  n < 1,  1 < s,  s < n \/ s = n,  i < n
 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (TRIV)
79)  n < 1,  1 = s,  s < n \/ s = n,  i < n
 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (TRIV)
80)  n < 1,  forall(i:Nat) i < 1 =>
 if A[i] = x then false else true fi,  i < n
 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (tif  81,84)
81)  n < 1,  forall(i:Nat) i < 1 =>
 if A[i] = x then false else true fi,  i < n
 ||-- A[i] = x => false,  1 < s,  1 = s (timpl  82)
82)  n < 1,
 forall(i:Nat) i < 1 => if A[i] = x then false else true fi,
 i < n,
 A[i] = x ||-- 1 < s,  1 = s (xrewrite  83)
83)  n < 1,
 forall(i:Nat) i < 1 => if A[i] = x then false else true fi,
 i < n,
 A[i] = x ||-- true,  1 = s (TRIV)
84)  n < 1,  forall(i:Nat) i < 1 =>
 if A[i] = x then false else true fi,  i < n
 ||-- ~ (A[i] = x) => true,  1 < s,  1 = s (xrewrite  85)
85)  n < 1,  forall(i:Nat) i < 1 =>
 if A[i] = x then false else true fi,
 i < n ||-- true,  1 = s (TRIV)
86)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s,  s < n \/ s = n (xrewrite  87)
87)  n < 1,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n ||-- if A[i] = x then false else true fi,
 true,  1 = s (TRIV)
88)  n < 1,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (xrewrite  89)
89)  n < 1,
```

```
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n ||-- true,  1 = s (TRIV)
90)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i = n ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (xrewrite  91)
91)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i = n ||-- 1 < s,  true (TRIV)
92)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n \/ i = n ||-- if A[i] = x then false else true fi,
 s < n \/ s = n (xrewrite  93)
93)  n < 1,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < 1 => if A[i] = x then false else true fi),
 i < n \/ i = n ||-- if A[i] = x then false else true fi,
 true (TRIV)
The proof consists of 93 nodes.


Q.E.D.
```

This proof was executed in 0.58 seconds on the aforementioned Ultra 1, while a (mostly) inductive proof with the command sequence mentioned above took 0.48 seconds.

5.4 Concluding Remarks

This concludes the tests of the ABEL proof system. We have seen that the verification proofs for even quite simple program code segments runs fairly long textually, but that the complexity of the expressions to be proved are in mostly their length, and hence they can often be proved with simple command sequences. We have also seen that even though deductive proofs with the `bpc` strategy generally runs textually longer, their execution time is comparable to those of inductive proofs. We have also seen that even with seemingly simple theorems like those discussed in this chapter, not all of them may be proved in a completely automatic fashion.

Chapter 6

Evaluation

In this chapter the previously described theory, examples and system tests are used to evaluate the ABEL system as it currently stands. First described are some concrete problems I have encountered. Following that is a discussion on how to avoid having to prove the same expression multiple times. Finally is a discussion on two topics that are of interest in developing the ABEL system into a verification system: strategies for further automating of proof construction, and how to discover programming errors in the code that is to be verified.

6.1 Problems in the ABEL System

During my testing of the system I identified a number of problems concerning the following:

BPC: Sometimes one wishes to perform a number of BPC steps, followed by induction. However, the bpc strategy will in this case also use the tall rule, removing the universal quantifier needed for induction. In addition, the non-constructive rules pose problems in incorporating them in strategies.

Induction: The problems I have found with the induction module mostly relates to the user interface, not the core induction facility. Nevertheless, finding remedies for these deficiencies would significantly help the usability of the system.

Existential quantifiers: I find existential quantifiers difficult to handle in the system in any way that does not rely completely on user invention. I will discuss whether the system may in any way do more in the handling of these.

These problems will be examined in the following, as well as some related topics.

6.1.1 The bpc Strategy

The BPC rule $T\forall$ may, if the bpc strategy is applied uncritically, remove a universal quantifier one later finds to be needed for induction. This is not strictly a problem pertaining to the bpc strategy, but rather with its interplay with the induction module.

I have identified two realistic remedies: either provide a generalisation operator, which binds a variable in a universal quantifier if possible as described in chapter 2; or extend the undo command to be able to revert parts of strategy applications—i.e. the offending tall application. As it now is, the user might have to use printproof to see the applied BPC rules, then undo the whole bpc application, and finally reapply the sequence of used rules up until tall before being able to perform the induction.

6.1.2 The Induction Module

The system currently only allows induction over the textually first occurring variable. Induction over any of the variables bound in an outer $\forall$ (or a sequence of $\forall$s, as that can be converted to a single $\forall$) should be allowed, as their ordering is arbitrary; either by the induction command accepting an argument of which variable to induct over, or by the system providing a command to rearrange the ordering of the variables bound by the $\forall$. The first of these alternatives seems to be the better in regard to usability.

Generalisation could perhaps be useful in situations in addition to the one mentioned above, involving the BPC tall rule. To avoid the problems with generalisation described in section 2.2.4, this facility could be coupled to the induction—in practice making the induction facility able to generalise variables if needed before performing the induction. Furthermore, induction should only be applied once to a variable in any one proof branch, so there should be a way to avoid generalising a variable that has already undergone induction.

6.1.3 The Existential Quantifier

One of the seemingly hardest things to prove without extensive reliance on the user is a theorem containing an existentially quantified variable. Constructing a proof for such a theorem often entails finding a value for the quantified variable for which the theorem is true, but simply searching sequentially for this value is hardly an acceptable strategy for types with infinite value space. Furthermore some such values, especially when the natural numbers are concerned, can seem intuitively obvious to the user, while being very hard to discover automatically.

The ABEL system has one operation only to deal with existential quantifiers, the BPC rule $T\exists$. This rule is not constructive: it makes the sequent

more complex, and hence must be treated with caution if included in a strategy. This rule takes as an optional argument a term to use for instantiation. Without such a term the system chooses a free variable for instantiation, but this rarely leads the proof any nearer success. Hence, the user is relied upon to supply a term for which the quantified expression is true. Because of this, the texist rule is not part of the bpc strategy.

Automating Existential Proof Construction?

As mentioned above, one obvious strategy for automated proof of theorems with existentially quantified variables is to search through the value space of the variable for a value for which the theorem can be proved. However, one soon realizes that this in most instances would entail the generation of a quite enormous amount of proof branches, as the further proof would have to be tried with each possible value until it is proved or the value space exhausted. A simple, linear search through the value space seems hardly to be ideal, unless the space is small—it could work well for the boolean type, with only two variables, and in some cases also for e.g. numbers (integers, natural numbers, etc.) if solutions involve low numbers (as it seems natural to start the search from 0), but it seems hard to make a general decision as to when, and for how long, one should try searching through the value space.

Another method is to try using the terms (of the correct type) already present in the sequent, as one of these often is the right one. However, this is not, evidently, a general strategy.

This seems to be one of the "interesting" areas of proof construction that are best left to the user; here the human intuition more than in most of the other parts of theorem proving comes into its right as inherently more effective than currently available computers.

The Relevance to Program Verification

When evaluating a system like the ABEL system—with the ultimate goal of being an environment for program verification—a timely question to ask when dealing with unresolved problems is, how often does this problem occur in actual use? If the existential quantifier appears often in realistic proof obligations we should naturally strive to make the system handle them better, but if it occurs only rarely, maybe the effort is better spent on other issues.

A review of a number of verification proofs—some given in various forms as exams assignments in the course IN 217 (Program Specification and Verification) at the Department of Informatics at the University of Oslo, others in Dahl's book [Dah92]—shows that existential quantifiers in fact do not appear very often in program verification. The proofs included in this

thesis is, I hope, also somewhat indicative; the only proof with an existential quantifier I have included can be found in Appendix B.1.

6.2 Recurring Subproofs

One way of improving the proving capabilities and speed of a theorem prover is to save proven sequents as rewrite rules (to true) in a database for later use when encountering this expression again. However, this is not completely trivial—the whole environment of assumptions, including loaded modules, will have to be saved in some way. If the loaded modules are saved as references, the system will need to be able to identify, upon deciding to use a saved sequent, if a module is changed to be inconsistent with the version in use when saving this sequent.

However, significant simplification of the proof process may be accomplished without those problems, if one keep the saving of proved expressions to within the proof under construction. In that case, the environment of loaded modules will be the same, and assumptions will also be somewhat easier dealt with.

As using these saved expressions involves searching for a matching expression, and may be viewed as rewrite rules from the expression to **true**, it seems the rewriter could be the place to put something like this. The rewriter already uses hashing to speed up matching of rules ([Mid99] p. 38), and this functionality might possibly be used to also search in saved theorems with no or only minor modifications.

6.2.1 What Should be Saved?

We are then left with the question of exactly what to save—there is the complete spectrum from only the starting theorem to every expression encountered during a (successful) proof to choose from. How many expressions to include is for a large part a question of efficiency in implementation. However, it seems at least the head expressions of proof branches should be included, as this is a middle way between all or just one; and as it seems a plausible that those expressions, and the branches they head, is what one would want to eliminate with this saving functionality.

6.2.2 Saving and Reusing Command Sequences

There is another possibility for tackling recurring expressions in proofs, which also could help proving of expressions that are similar to earlier proved expressions, but which the rewriter nevertheless fails to match. This facility would make it possible to tell the system to apply the same proof sequence as that which proved an earlier completed proof branch.

6.3 Advanced Strategies

As described in section 2.2.6 in chapter 2, the strategy mechanism implemented in the ABEL system supports well the relatively straightforward command sequences required to complete the proofs of chapter 4. Here, I will discuss how we can construct more advanced strategies, with the aims of both automating more of the proof construction process, and constructing more general tools in the hope of requiring less user proficiency in theorem proving. More to the point, we want to investigate the possibility of constructing a strategy that makes a "sensible" choice between using BPC and induction, to complete the proofs of chapters 4 and 5 without further user intervention.

Induction seems to give shorter proofs more often, so we might try

```
repeat [xrewrite, induct +, bpc]
```

to use induction as far as possible (the plus sign signifies any one part of the consequent), and only when induction fails try BPC. This will obviously complete many proofs, including all where rewriting and induction alone suffice. It does however not seem obvious to me from the examples that this will accomplish much more than the same without bpc.

```
[repeat [xrewrite, induct +], undo, repeat [xrewrite, bpc]]
```

This command sequence includes a simple case of rollback—if the inductive strategy does not work we try BPC reasoning instead. However, this strategy is not significantly more powerful than separate strategies for inductive and BPC-based proofs; many proofs require a combination of induction and BPC, not just a choice of either one exclusively. But if we just apply repeat to the above command we will go into an infinite rule as soon as the inductive proof attempt fails, forever undoing and redoing the failing induction.

What we need is a means to revert to some specific place in the proof tree and continue execution from there. This is not supported in the current command language; I will address this below.

6.3.1 How to Choose the Right Rule

At some points in any non-trivial proof there is more than one rule that can be applied. The way a strategy is programmed decides which path to take, but what if this choice leads to a dead end, where another rule would lead to success? An example I have mentioned before is the BPC rule T∀, which instantiates a universal quantifier in the consequent—where induction also might be applied. Sometimes use of the bpc strategy leads to a fairly long row of BPC rules, with T∀ at or near the end. If then the proof branch fails

and the user want to try induction instead of T∀, one has to use printproof to see the BPC rules applied, use undo or prune to undo the whole row of BPC rules applied, and then apply all the desired BPC rules by hand up until T∀. There are several possible solutions to this; I will explore two in the following.

6.3.2 Partial Undo

The first, and simpler solution is to extend the undo command to be able to revert just some of the steps of a previously applied strategy. This would necessitate some means of identifying the point in the proof at which one would wish to stop the reverting—one possible method is to modify printproof to give each step identifying marks the user could provide as an argument to undo. Implementing this seems conceptually trivial, so I will not discuss it further.

6.3.3 Automated Rollback

The second possible solution I will discuss is to make strategies that tries to revert back to the most likely split point—a generalisation of the above attempt to first try proof through induction, then through BPC.

As the only way to find if a branch of a proof will lead to success is to try it out, an automatic theorem prover needs the ability to go back to points earlier in the proof tree where choice were made between more than one applicable rule. Now, as the present ABEL system is an interactive proof helper, not an automatic prover, we do not need a fully successful rollback mechanism; however, as the aim of strategies is to lessen the work of the user of the system, it *is* a goal to make proof construction as automatic as reasonably possible, or at least provide the means for the user to automate. I.e. we want to provide a strategy mechanism that supports automating of proofs as much as possible.

Let us examine some possible strategies with rollback. The simplest, and most obvious, is to try all the rules (in some ordering, which would need to take heed of things like the non-constructive BPC rules), and use undo if none work:

repeat [all rules in some ordering, undo]

However, unless some nondeterminism was built into the order of application of the rules this strategy is bound to loop endlessly, as the last successful rule would just be reverted and reapplied endlessly. Additionally, the undo would only revert the last proof step, while the branching point we need to reach very well could be further back. Obviously, we need something more elaborate, making sure not to reapply reverted rules.

Let us examine an attempt at a strategy for proofs using both induction and BPC reasoning, and incorporating undo to repeatedly try the other when one fails:

```
repeat [xrewrite, induct 1], undo, repeat [xrewrite, bpc]
```

The reasoning behind this command sequence is first to try inductive proof, then BPC reasoning if inductive does not succeed. This command sequence will work, however there is no repetition at the top level, so it will not greatly reduce the work load for the user. To be really useful such a command sequence would have to step back one induction application at a time and try the bpc strategy at each intermediary step. Conversely the sequence should do the same when BPC fails, going back one step at a time and try induction.

6.3.4 A Rollback Mechanism Proposal

In strategy programming the purpose of a rollback mechanism must be to go back to a certain command in a proof and make that command fail, so as to continue with the next command in a square bracket list (which, we recall, is a list of alternatives tried in order until one succeeds). It is conceivable that this can be be accomplished by having a marking operator, which puts a mark in the proof tree or something similar, and extending the undo (or possibly prune) command to take as argument such a mark, and make the command associated with that mark fail. An example follows:

```
[m1 : [c1, undo m1], c2]
```

Here c1 would be a command sequence. If c1 fails to complete the proof undo would prune off this proof branch and instead try c2.

This kind of control is crucial to the development of more automatic strategies, as I have experienced that no simple strategy without rollback could complete all the proofs in this thesis.

6.4 How to Discover Errors

As mentioned earlier, the ABEL system is not meant as a theorem prover for mathematical theorems, but rather for verification of programming code. This is an important distinction: mathematicians generally only attempt to prove theorems they believe are true, while verification systems should try to find, and point to as accurately as possible, errors in program code. Hence the prover part of a verification system should be able to in some way proclaim an expression false, or at least unprovable, and preferably do

this in a way that is as helpful as possible for identifying the unverifiable code.

The issue of when to stop and proclaim that an expression is not a theorem is extremely difficult; in fact it is undecidable, as can be shown from the theory of incompleteness of axiomatic systems—a consequence of Gödel's theorem of incompleteness is that there exists theorems that cannot be proven to be true. As the system does not have built-in support for *dis*-proving an expression, we can, at least in principle, never say that an expression is not a theorem, but only that the system is unable to complete a proof for the expression.

Nevertheless, I will in the following explore what error-detection capabilities may be expected in a program verification system.

6.4.1 Proof Branch Exhaustion

If the prover exhausts all possible proof branches, that is, all applicable rules at all possible points in the proof tree have been tried, the system can pronounce to be unable to prove the given expression. This does not change if rules that accept user input like $T\exists$ are applicable at some point; it should be the user's responsibility to explore possible applications of such rules.

Proof branch exhaustion can obviously only be expected to occur where there is a lessening of complexity in the theorems; the constructive BPC rules are such a system, while ND are not; proof systems relying on ND or similar deduction systems can generally never expect to exhaust all proof branches.

6.4.2 Loops

If using non-constructive rules one can get loops in the proof, where the non-constructive rules introduce complexity, while constructive rules removes it again, bringing the proof back to an earlier encountered expression. This could be detected (although the trivial method of saving all expressions and searching for matches might have a performance impact), but what the proper handling of this is does not seem obvious to me—as the non-constructive rules mostly require user insight to lead the proof nearer completion, the user should probably be consulted as to whether to fail the proof upon discovery of loops.

6.4.3 Other Errors

Other errors, many of a semantic nature (syntax errors should be handled by the parser, and never reach the prover) should also be handled in a way that supports identification of the offending program code. One example

is incompletely specified functions—a function f defined by generator induction with the `case` construct is not required to handle all generators. If f then is called in the program code to handle a generator for which it is not defined, the system should be able to identify this as the actual problem.

6.4.4 What to Do When Failing

An unproved proof obligation implies that the proof for some part of a program fails, and this program code has to be rewritten. A proof obligation generator doing Hoare analysis is not implemented in the current ABEL system, but it seems unlikely that it should pose serious problems having such a generator do proof obligation generation in a way that allows tracking back from a failed proof obligation to the program code section the obligation was generated for. If e.g. the entrance proof obligation for a loop fails, this gives the user relatively good information on what has to be changed.

As we have seen, in practical verification we have to treat unprovable expressions as faults. That is, we should not consider what is in reality theorems, just what we (or rather our system) can prove.

Chapter 7

Conclusions

We have seen that the ABEL system, as it currently stands, seems able to prove some realistic verification proof obligations, without requiring deep understanding of theorem proving on the user's part. The proofs are quite lengthy, even from relatively simple program code, but may often be proved with rather simple command combinations. We can therefore conclude that the ABEL system's prover functions well as a proof checker and proof construction helper, and could probably with relatively few additions and modification be used in a system for effective, real-life program verification.

We have seen some proposals on how to improve the user's proof construction efficiency with a system such as the ABEL system. Most of these proposals deal with further automating of the proof construction process, to reduce the system's reliance on user guidance.

We have seen that a means to incorporate rollback into command sequences would enhance the power of the strategy mechanism, making it easier to program strategies that could fully automate the construction of all the example proofs we have examined in this thesis.

The tests presented has shown us that the theorems generated from program verification for the most part may easily be handled with at most a few different high-level strategies for repeated application of induction or BPC rules (or combinations thereof). However, more advanced theorems do indeed occur, and require adeptness at proving in the user.

7.1 The Current System

The ABEL system as it now stands supports, as we have seen, theorem proving quite well, and the parts presently implemented would not need much augmentation to be used in a verification system. However, the system is presently no more than a theorem prover, and a few larger subsystems will have to be implemented to make it a true verification system.

As shown in chapter 5, verification proofs with the system often run rather long in the number of steps, even for quite simple proof obligations. However, the proofs may often be completed with simple strategies, and does not take very long to execute. Hence, it seems viable in practical verification to let the system try a few simpler strategies, and only if none of them succeed prompt the user for guidance.

7.1.1 What is Missing

While proving of theorems is, as we have seen, quite well supported by the current ABEL system, what is still missing is the ability to process program source code and generate verification proof obligations. While the system has a parser and type-checker for the applicative parts of the ABEL language, it lacks support for the imperative parts. Neither is functionality implemented for the processing of Hoare sentences. (Hoare analysis was implemented in older versions of the system, but is not (yet) in the current.) In order to become a full-fledged verification tool the system will have to be extended in both these areas.

Additionally, there should be functionality in the prover for failing a proof—at minimum the user should be able to declare a proof a failure, but it would be preferable if the system could recognise failures as described in section 6.4. The system should then be able to trace a failed proof obligation back to the source code for which the obligation was generated.

7.1.2 What Could be Improved

We saw in section 6.3 that with some amendment, the command language could support construction of strategies with more detailed control of operation at failure. This would seemingly facilitate strategies that could autonomously complete proofs for many simpler theorems where the proof cannot be completed by induction or deduction alone. We have seen that the proof obligations generated from verification of programs are often somewhat large, but with a simple structure, requiring long, yet relatively simple and tedious proofs. Even though the command sequences for the shown proofs are not very long in isolation, one has to keep in mind that the program code those were generated for were very simple, and that a realistic program would contain thousands of such, and more complex, parts. Hence, near complete automation of proving such theorems would be desirable in a verification system. Nevertheless, the system should still be interactive, as there will be occasions when the user will have to help.

Furthermore, we have seen that an amendment to the `undo` command, to make it possible to revert parts of an applied command sequence or strategy (i.e. the last step(s) listed in `printproof`), would help usability in

that it would allow reverting of an application of tall without reverting the whole application of the bpc (or other) strategy it was part of.

Finally, we have seen that the induction facility of the current system should be extended with an optional argument of the variable to be used for the induction. Presently only induction on the textually first variable in a universal quantifier is possible; while induction on the first variable does of course not always lead to the shortest proof. However, it is not sufficient to use the tall rule to remove the quantified variables preceding the variable one wants to use for induction, as one could very well need to apply induction to these outer variables at a later stage.

7.1.3 A Note on Development of Software

The current implementation of the ABEL project has grown to a size where I find it very difficult to get to know the inside workings of the system. One of the main reasons for this is a total lack of documentation. Even if the various parts of the system (and, most importantly, their interfaces) are well designed, the lack of documentation makes it very difficult to understand how to use the system to implement new functionality.

Bastiansen writes in [Bas95] that one hopes the modularisation properties of Standard ML are powerful enough. I think they proved adequate for the ABEL system; the most problems I have had with the code, besides the lack of documentation, is its sheer complexity and size—coupled with, among other things, the unfortunate ML tradition of using variable names of no more than three characters, which I think is not a good practice in functions of several hundred lines.

7.2 The Road Ahead

If further development of the system is undertaken, there is several starting points, as described above. A proof obligation generator implementing processing of Hoare logic is probably a relatively free-standing task for one or two cand.scient. theses. Implementing support for the imperative parts of the ABEL language should also be a reasonable undertakement.

However, it is not obvious to me that this line of research, i.e. verification of a language resembling the traditional ALGOL family, is a goal worthy of further pursuit. IN 217, the course to which the ABEL project has traditionally been (more or less) tightly coupled, and in which the system has occasionally been used, has now been changed (and has changed name to INF 220, Formal Modelling and Execution of Communicating Processes) to teach formal modelling with Maude [CDE$^+$]. Additionally, in general the research in verification has shifted towards formal specification languages (which are refined down to executable code). Those specification

languages are either of a middle level, like the PVS language, where serious theorem proving is still needed; or designed to be very high-level and easy to prove correct, like the B method ([Wor96], [bco]) and TLA+ ([Lam]).

The course IN 305 (Parallel programming and operating systems) uses Hoare logic to verify parallel programs, and a possible direction of further development of the ABEL system would be to implement a system for use in this course.

However, I think a natural continuation of the ABEL project, and, I believe, an interesting line of research, would be to let the system evolve with the INF 220 course. The ABEL project has drawn on IN 217 in the past in a way that has been beneficiary for both the project and the course, and I believe it would likewise be beneficiary for a future project to draw on INF 220. Unless INF 220 is modified, that would mean the construction of a system for a language similar to Maude.

Bibliography

[abe] The ABEL project's home page. http://www.ifi.uio.no/ prover/abel/.

[arg] The homepage for automated reasoning at Argonne. http://www-unix.mcs.anl.gov/AR/.

[Bas95] Tore Jahn Bastiansen. Modulsammensetning og semantisk analyse i ABEL. Cand.scient. thesis, Institutt for informatikk, Universitetet i Oslo, http://www.ifi.uio.no/~prover/abel/, 1995.

[bco] B-Core Ltd. http://www.b-core.com/.

[BKR92] A. Bouhoula, E. Kounalis, and M. Rusinowitch. SPIKE: A theorem-prover. In *Proceedings of LPAR'92, Lecture Notes in Artificial Intelligence 624*. Springer-Verlag, 1992.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.

[Bre98] Olav Andree Brevik. En interaktiv bevishjelper. Cand.scient. thesis, Institutt for informatikk, Universitetet i Oslo, http://www.ifi.uio.no/~prover/abel/, 1998.

[CDE+] Manuel Clavel, Francisco Durdán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude manual. Published online. http://maude.csl.sri.com/manual/maude-manual-html/.

[Dah92] Ole-Johan Dahl. *Verifiable Programming*. Prentice Hall, 1992.

[Gö31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[Hof79] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books Inc., 1979.

[Kir] Bjørn Kirkerud. Notes on rewriting to the course in 307. Published online. http://www.ifi.uio.no/in307/notater/.

[Kor98] Leif John Korshavn. Termomskrivingsregler i ABEL. Cand.scient. thesis, Institutt for informatikk, Universitetet i Oslo, http://www.ifi.uio.no/~prover/abel/, 1998.

[Lam] Leslie Lamport. Assorted TLA+ literature. Published online. http://research.microsoft.com/users/lamport/tla/tla.html.

[Lin99] Ole Christian Lingjære. Introduksjon til Hoare-logikk og abstrakte typer, 1999. Kompendium 55.

[Lus92] Ewing L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. Springer LNAI #624, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA, 1992. http://www.mcs.anl.gov/home/mccune/ar/lisk-lpar.ps.

[Mid99] Steinar Midtskogen. Termomskriving i ABEL. Cand.scient. thesis, Institutt for informatikk, Universitetet i Oslo, http://www.ifi.uio.no/~prover/abel/, 1999.

[MS95] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, April 1995. ieeecs. http://www.csl.sri.com/papers/wift95/.

[NN58] Ernest Nagel and James R. Newman. *Gödel's Proof*. Routledge, 1958.

[RSS99] H. Rueß, N. Shankar, , and M.K. Srivas. Modular verification of SRT division. *Formal Methods in Systems Design*, 14(1):45–73, jan 1999. http://www.csl.sri.com/papers/fmsd99/.

[Sha96a] N. Shankar. PVS: combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 257–264, Palo Alto, CA, nov 1996. Springer-Verlag. http://www.csl.sri.com/papers/fmcad96-shankar/.

[Sha96b] Natarajan Shankar. Unifying verification paradigms. In Bengt Jonsson and Joachim Parrow, editors, *Formal*

Techniques in Real-Time and Fault-Tolerant Systems, volume 1135 of *Lecture Notes in Computer Science*, pages 22–39, Uppsala, Sweden, sep 1996. Springer-Verlag. http://www.csl.sri.com/papers/ftrtft96/.

[SORSC99a] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Ref*. SRI International, http://pvs.csl.sri.com/, version 2.3 edition, 1999.

[SORSC99b] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. SRI International, http://pvs.csl.sri.com/, version 2.3 edition, 1999.

[Wal96] Toby Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

[Wor96] J.B. Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.

Appendix A

Standard Modules from the ABEL System

Here are two standard modules from the ABEL system which are used in the examples in this thesis, *Int* and *Seq*.

A.1 The Int Module

The Int module implements the integer type, and some functions.

```
Int ==
module
  (* The integer type *)
  type Int by Neg, Zero, Pos
       where NPos = Neg + Zero,
             NZro = Neg + Pos,
             Nat = Zero + Pos

  func Z : Zero
  func S : Nat ⟶ Pos
  func N : Pos ⟶ Neg

  oneone genbas Int == Z, S, N

  (* Producers *)
  func succ : Int ⟶ Int
  func pred : Int ⟶ Int
  func neg : Int ⟶ Int
  func abs : Int ⟶ Nat
  func ^+^ : Int * Int ⟶ Int
```

```
func ^−^ : Int * Int ⟶ Int
func ^*^ : Int * Int ⟶ Int
func ^_mod_^ : Int * Pos ⟶ Nat
func ^/^ : Int * NZro ⟶ Int

(* Observers *)
func ^≤^ : Int * Int ⟶ Bool
func ^<^ : Int * Int ⟶ Bool
func ^≥^ : Int * Int ⟶ Bool
func ^>^ : Int * Int ⟶ Bool

def succ(x) == case x of
                 Z ⟶ S(Z)
               | S(x') ⟶ S(S(x'))
               | N(S(Z)) ⟶ Z
               | N(S(S(x'))) ⟶ N(S(x'))
               fo

def pred(x) == case x of Z ⟶ N(S(Z))
             | S(x') ⟶ x' | N(x') ⟶ N(S(x')) fo

def neg x == case x of Z ⟶ Z | S(x) ⟶ N(S(x)) | N(x) ⟶ x fo

def abs(x) == case x of Z ⟶ Z | S(x) ⟶ S(x) | N(x) ⟶ x fo

def x + y == case x of
               Z ⟶ y
             | S(x') ⟶ case y of
                         Z ⟶ x
                       | S(y') ⟶ S(S((x'+y') qua Nat ))
                       | N(y') ⟶ x−y'
                       fo
             | N(x') ⟶ case y of
                         Z ⟶ x
                       | S(y') ⟶ y−x'
                       | N(y') ⟶ N((x'+y') qua Pos)
                       fo
             fo

def x − y == case x of
               Z ⟶ neg y
             | S(x') ⟶ case y of
                         Z ⟶ S(x')
                       | S(y') ⟶ x'−y'
```

```
                        | N(y') ⟶ x+y'
                        fo
                | N(x') ⟶ case y of
                            Z ⟶ x
                          | S(y') ⟶ pred(x−y')
                          | N(y') ⟶ x+y'
                          fo
                fo

def x * y == case y of
               Z ⟶ Z
             | S(y') ⟶ (x*y')+x
             | N(y') ⟶ neg(x*y')
             fo

def x _mod_ y == case y of S(y) ⟶ case x of
                                    Z ⟶ Z
                                  | S(x') ⟶ let v == x' _mod_ S y in
                                            if v = y then Z else S(v) fi ni
                                  | N(x') ⟶ let v == x' _mod_ S y in
                                            if v = Z then Z else S(y − v) fi ni
                                  fo fo

def x / y == case y of
               S(y') ⟶
                 case x of
                   Z ⟶ Z
                 | S(x') ⟶
                     let v == x' / y in
                      if (x' _mod_ y) = y' then S v else v fi ni
                 | N(x') ⟶ neg(x'/y)
                 fo
             | N(y') ⟶ neg(x / y')
             fo

lemmas(x,y:Int) (x+y)=(y+x)

def x < y ==
  case x of
       Z ⟶ case y of N x' ⟶ false
                   | S(x') ⟶ true | Z ⟶ false fo
     | S(x') ⟶ case y of S y' ⟶ x' < y'
                   | N y' ⟶ false | Z ⟶ false fo
     | N(x') ⟶ case y of N y' ⟶ y' < x'
```

```
                        | S y'  ⟶ true | Z ⟶ true fo
    fo

  def x ≤ y == (x < y) \/ (x = y)
  def x ≥ y == y ≤ x
  def x > y == y < x
endmodule
```

A.2 The Seq Module

The Seq module implements the sequence type, and some functions.

```
Seq ==
module
  include Int

  typevar T

  type Seq{T} by ESeq, NESeq

  func e : ESeq{T}
  func ^⊢^ : Seq{T} * T ⟶ NESeq{T}

  oneone genbas Seq == e, ^⊢^

  func ^ −| ^ : T * Seq{T} ⟶ NESeq{T}
  func ^ ⊢| ^ : Seq{T} * Seq{T} ⟶ Seq{T}
  func # : Seq{T} ⟶ Nat
  func rt : NESeq{T} ⟶ T
  func lr : NESeq{T} ⟶ Seq{T}
  func lt : NESeq{T} ⟶ T
  func rr : NESeq{T} ⟶ Seq{T}

  def x −| q == case q of e ⟶ e ⊢ x | q' ⊢ y ⟶ (x −| q') ⊢ y fo
  def q ⊢| r == case r of e ⟶ q | r' ⊢ x ⟶ (q ⊢| r') ⊢ x fo
  def # q == case q of e ⟶ Z | q ⊢ x ⟶ S(#q) fo
  def rt(q ⊢ x) == x
  def lr(q ⊢ x) == q
  def lt(q ⊢ x) == case q of e ⟶ x | q'⊢x ⟶ lt(q) fo
  def rr(q ⊢ x) == case q of e ⟶ e | q'⊢ _ ⟶ rr(q) ⊢ x fo

  func ^_has_^ : Seq{T} * T ⟶ Bool
```

```
func norep : Seq{T} ⟶ Bool
func sub : Seq{T} * Seq{T} ⟶ Bool
func head : Seq{T} * Seq{T} ⟶ Bool
func tail : Seq{T} * Seq{T} ⟶ Bool
func segm : Seq{T} ⟶ Seq{T} ⟶ Bool

def q _has_ x == case q of e ⟶ false
                       | q' ⊢ y ⟶ (x = y) \/ (q' _has_ x) fo
def norep(q) == case q of e ⟶ true
                        | q' ⊢ x ⟶ ¬(q' _has_ x) ∧ norep(q')
                fo
def sub(q,r) == case q of e ⟶ true | q' ⊢ x ⟶
                  case r of e ⟶ false | r' ⊢ y ⟶
                    if x = y then sub(q',r') else sub(q,r') fi fo fo
def head(q,r) == (q = r) \/ case r of e ⟶ false
                                 | r' ⊢ x ⟶ head(q,r') fo
def tail(q,r) == case q of e ⟶ true | q' ⊢ x ⟶
                   case r of e ⟶ false | r' ⊢ y ⟶
                     (x = y) \/ tail(q',r') fo fo
def segm q r == case r of e ⟶ q = e | r' ⊢ x ⟶
                  (tail(q,r)) \/ (segm q r') fo


func{T,U} map: (T ⟶ U) ⟶ Seq{T} ⟶ Seq{U}
def map f q == case q of e ⟶ e | q⊢x ⟶ map f q ⊢ f x fo

func{T,U} foldl,foldr: (U * T ⟶ U) ⟶ U ⟶ Seq{T} ⟶ U
def foldl f x q == case q of e ⟶ x | r ⊢ y ⟶ f(foldl f x r,y) fo
def foldr f x q == case q of e ⟶ x | r ⊢ y ⟶ foldr f (f(x,y)) r fo

func tabulate : (Nat * (Int ⟶ T)) ⟶ Seq{T}
def tabulate (n,f) == case n of Z ⟶ e | S n' ⟶ tabulate(n',f)⊢f n' fo

func ^[^] : NESeq{T} * Pos ⟶ T
def (q⊢x)[S i] == if i= #q then x else q[S i] fi

func sb : NESeq{T} * Pos * Pos ⟶ T
def sb(q⊢x,S i,l) == if i = 1 then x else sb(q,S i,(pred l) as Pos) fi

func ^[^−>^] : NESeq{T} * Pos * T ⟶ NESeq{T}
def (q⊢x)[S i −> y] ==
      if i = #q then q⊢y else (q as NESeq{T})[S i −> y] ⊢ x fi

func  ^[^..^] : Seq{T} * Pos * Nat ⟶ Seq{T}
```

```
  def q[i..j] ==
        case q of e ⟶ e
                  | q⊢x ⟶ if i>j then e
                          else if j > #q then q[i..(pred j) qua Nat]⊢x
                          else q[i..(pred j) qua Nat] fi fi fo

  (* Higher order *)

  func sort : (T * T ⟶ Bool) ⟶ Seq{T} ⟶ Seq{T}
  func ins : (T * T ⟶ Bool) ⟶ Seq{T} * T ⟶ Seq{T}

  def ins less (q,x) == case q of
                          e ⟶ e⊢x
                        | q'⊢y ⟶ if less(x,y) then
                                   ins less (q',x) ⊢ y
                                 else q⊢x fi fo

  def sort less q == case q of e ⟶ e
                               | q⊢x ⟶ ins less (sort less q,x) fo

endmodule
```

Appendix B

Additional Proofs

B.1 Proof of an Existentially Quantified Variable

The following is a proof of a theorem I found in the IN 217 exam from 1995. It demonstrates proving a theorem containing an existentially quantified variable. I have included the whole proof transcript to show a further example, in addition to those in chapter 3, of how a proof session with the ABEL system might typically be conducted. The theorem to be proved is:

$$\forall(q : \text{Seq\{T\}}, x : \text{Int})\exists(i : \text{Nat})((1 \leq i) \wedge (i \leq \#q) \wedge x = q[i]) = (x \in \text{set}(q))$$

```
ABEL> prove forall(q:Seq{Int},x:Int) exist(i:Nat)
 ((1 <= i) /\ (i <= # q) /\ x = q[i]) = (x _in_ set(q))

1:

 True (empty)
|-----
# 1) forall(q:Seq{Int},x:Int) (exist(i:Nat)
     (1 <= i /\ i <= # q /\ x = q[i] <=> x _in_ set q))

PROVER> xrewrite

1:

 True (empty)
|-----
# 1) forall(q:Seq{Int},x:Int) (exist(i:Nat)
     ((1 < i \/ 1 = i) /\ (i < # q \/
       i = # q) /\ x = q[i] <=> x _in_ set q))

PROVER> induct 1
This yields 2 subgoals:
```

```
1.1:

 True (empty)
|-----
# 1) forall(x:Int) (exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < # e \/ i = # e) /\ x = e[i] <=> x _in_ set e))


1.2:

#-1) forall(x:Int) (exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < # q \/ i = # q) /\ x = q[i] <=> x _in_ set q))
|-----
# 1) forall(x:Int) (exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < # (q |- q'el) \/ i = # (q |- q'el)) /\
     x = (q |- q'el)[i] <=> x _in_ set (q |- q'el)))



1.1:

 True (empty)
|-----
# 1) forall(x:Int) (exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < # e \/ i = # e) /\ x = e[i] <=> x _in_ set e))

PROVER> xrewrite

1.1:

 True (empty)
|-----
# 1) forall(x:Int) (exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ x = e[i] <=> false))

PROVER> induct 1
This yields 3 subgoals:

1.1.1:

 True (empty)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ 0 = e[i] <=> false)


1.1.2:

#-1) exist(i:Nat) ((1 < i \/ 1 = i) /\
```

```
     (i < 0 \/ i = 0) /\ x = e[i] <=> false)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ S x = e[i] <=> false)


1.1.3:

#-1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ x = e[i] <=> false)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ N x = e[i] <=> false)



1.1.1:

 True (empty)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ 0 = e[i] <=> false)

PROVER> xrewrite
Gives no changes.

PROVER> texist 1

1.1.1:

 True (empty)
|-----
# 1) ((1 < i \/ 1 = i) /\ (i < 0 \/ i = 0) /\
     0 = e[i] <=> false) \/ (exist(i:Nat) ((1 < i \/ 1 = i)
     /\ (i < 0 \/ i = 0) /\ 0 = e[i] <=> false))

PROVER> xrewrite
Gives no changes.

PROVER> bpc
This yields 4 subgoals:

1.1.1.1.1.1:

#-1) 1 < i, #-2) i < 0, #-3) 0 = e[i]
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ 0 = e[i] <=> false)
```

```
PROVER> xrewrite

1.1.1.1.1.1:

#-1) 1 < i, #-2) i < 0, #-3) 0 = e[i]
|-----
# 1) false

PROVER> undo
Undoing the last proof-step.


1.1.1.1.1.1:

#-1) 1 < i, #-2) i < 0, #-3) 0 = e[i]
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ 0 = e[i] <=> false)

PROVER> undo
Undoing the last proof-step.


1.1.1:

 True (empty)
|-----
# 1) ((1 < i \/ 1 = i) /\ (i < 0 \/ i = 0) /\
     0 = e[i] <=> false) \/
     (exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ 0 = e[i] <=> false))

PROVER> tor 1

1.1.1:

 True (empty)
|-----
# 1) (1 < i \/ 1 = i) /\ (i < 0 \/ i = 0) /\
     0 = e[i] <=> false,
# 2) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ 0 = e[i] <=> false)

PROVER> xrewrite
Gives no changes.

PROVER> undo
Undoing the last proof-step.
```

```
1.1.1:

 True (empty)
|-----
# 1) ((1 < i \/ 1 = i) /\ (i < 0 \/ i = 0) /\
     0 = e[i] <=> false) \/
     (exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ 0 = e[i] <=> false))

PROVER> undo
Undoing the last proof-step.


1.1.1:

 True (empty)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ 0 = e[i] <=> false)

PROVER> texist 1 ‘2‘

1.1.1:

 True (empty)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ 0 = e[i] <=> false),
# 2) (1 < 2 \/ 1 = 2) /\ (2 < 0 \/ 2 = 0) /\
     0 = e[2] <=> false

PROVER> xrewrite

1.1.1:

 True (empty)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ 0 = e[i] <=> false),
# 2) true
Which is trivially true!



Changing current to ’1.1.2’.

1.1.2:
```

```
#-1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ x = e[i] <=> false)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ S x = e[i] <=> false)

PROVER> xrewrite

1.1.2:

#-1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ x = e[i] <=> false)
|-----
# 1) true
Which is trivially true!



Changing current to '1.1.3'.

1.1.3:

#-1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ x = e[i] <=> false)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ N x = e[i] <=> false)

PROVER> texist 1 '2'

1.1.3:

#-1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ x = e[i] <=> false)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ N x = e[i] <=> false),
# 2) (1 < 2 \/ 1 = 2) /\ (2 < 0 \/ 2 = 0) /\
     N x = e[2] <=> false

PROVER> xrewrite

1.1.3:

#-1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ x = e[i] <=> false)
|-----
# 1) exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < 0 \/ i = 0) /\ N x = e[i] <=> false),
```

```
# 2) true
Which is trivially true!



Changing current to '1.2'.

1.2:

#-1) forall(x:Int) (exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < # q \/ i = # q) /\ x = q[i] <=> x _in_ set q))
|-----
# 1) forall(x:Int) (exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < # (q |- q'el) \/ i = # (q |- q'el)) /\
     x = (q |- q'el)[i] <=> x _in_ set (q |- q'el)))

PROVER> xrewrite

1.2:

#-1) forall(x:Int) (exist(i:Nat) ((1 < i \/ 1 = i) /\
     (i < # q \/ i = # q) /\ x = q[i] <=> x _in_ set q))
|-----
# 1) true
Which is trivially true!

The proof consists of 14 nodes.


Q.E.D.
```

We see that overall this is an inductive proof; sequent 1 is split into 1.1 and 1.2 by induction. Furthermore we see in branch 1.1.1 that several attempts is made using different techniques before the branch succeeds after application of texist with substitution of '2' for the existentially quantified variable. This substitution is also used in the application of texist in branch 1.2.

This proof can be performed quickly if one knows what to do; execution time on a modern computer is insignificant compared to the time the user needs to figure out the next step.

B.2 A lengthy BPC proof

Here I show a proof that runs rather long; it shows that the bpc strategy can produce long proofs with relatively little work for the user. The theorem is from Example 29 in [Lin99], and is

$$\forall(k, n, s, x : \text{Nat}, A : \text{Seq\{T\}})1 \leq k \wedge k < n \wedge$$
$$\textbf{if } 1 \leq s \wedge s \leq n \textbf{ then } A[s] = x \textbf{ else } \forall(i : \text{Nat})i \leq k \Rightarrow A[i] \neq x \textbf{ fi} \Rightarrow$$
$$\textbf{if } 1 \leq s \wedge s \leq n \textbf{ then } A[s] = x \textbf{ else } \forall(i : \text{Nat})i < k + 1 \Rightarrow A[i] \neq x \textbf{ fi}$$

```
1)  ||-- forall(k:Nat,n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 1 <= k /\ k < n /\
 if 1 <= s /\ s <= n
   then A[s] = x
   else forall(i:Nat) i <= k => A[i] =/= x
 fi =>
 if 1 <= s /\ s <= n
   then A[s] = x
   else forall(i:Nat) i < k + 1 => A[i] =/= x
 fi (xrewrite  2)
2)  ||-- forall(k:Nat,n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (tall  3)
3)  ||-- forall(n:Nat,s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (tall  4)
4)  ||-- forall(s:Nat,A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
```

```
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (tall  5)
5)  ||-- forall(A:Seq{Nat},x:Nat)
 (1 < k \/ 1 = k) /\ k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (tall  6)
6)  ||-- forall(x:Nat) (1 < k \/ 1 = k) /\ k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (tall  7)
7)  ||-- (1 < k \/ 1 = k) /\ k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi =>
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (timpl  8)
8)  (1 < k \/ 1 = k) /\ k < n /\
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (aand  9)
9)  (1 < k \/ 1 = k) /\ k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
```

```
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (aand  10)
10)  1 < k \/ 1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (aor  11,94)
11)  1 < k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (tif  12,65)
12)  1 < k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- (1 < s \/ 1 = s) /\ (s < n \/ s = n) =>
 A[s] = x (timpl  13)
13)  1 < k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n)
 ||-- A[s] = x (aand  14)
14)  1 < k,
 k < n,
```

```
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 < s \/ 1 = s,
 s < n \/ s = n ||-- A[s] = x (aor  15,40)
15)  1 < k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 < s,
 s < n \/ s = n ||-- A[s] = x (aor  16,28)
16)  1 < k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 < s,
 s < n ||-- A[s] = x (aif  17)
17)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i =
 k => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x (aimpl  18,27)
18)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i =
 k => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x (anot  19)
19)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i =
 k => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  (1 < s \/ 1 = s) /\ (s < n \/ s = n)
 (tand  20,22)
```

```
20)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i =
 k => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  1 < s \/ 1 = s (tor  21)
21)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i =
 k => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
22)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i =
 k => if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  s < n \/ s = n (aimpl  23,25)
23)  1 < k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,
 s < n ||-- A[s] = x,  s < n \/ s = n (tor  24)
24)  1 < k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,
 s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
25)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 < s,
 s < n ||-- A[s] = x,  s < n \/ s = n (tor  26)
26)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 < s,
 s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
27)  1 < k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
```

```
 1 < s,
 s < n ||-- A[s] = x (TRIV)
28)  1 < k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 < s,
 s = n ||-- A[s] = x (aif  29)
29)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x (aimpl  30,39)
30)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x (anot  31)
31)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  (1 < s \/ 1 = s) /\
 (s < n \/ s = n) (tand  32,34)
32)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  1 < s \/ 1 = s (tor  33)
33)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
```

```
34)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  s < n \/ s = n (aimpl  35,37)
35)  1 < k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,
 s = n ||-- A[s] = x,  s < n \/ s = n (tor  36)
36)  1 < k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,
 s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
37)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 < s,
 s = n ||-- A[s] = x,  s < n \/ s = n (tor  38)
38)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 < s,
 s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
39)  1 < k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x (TRIV)
40)  1 < k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 = s,
 s < n \/ s = n ||-- A[s] = x (aor  41,53)
41)  1 < k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
```

```
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 = s,
 s < n ||-- A[s] = x (aif  42)
42)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x (aimpl  43,52)
43)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x (anot  44)
44)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x,  (1 < s \/ 1 = s) /\
 (s < n \/ s = n) (tand  45,47)
45)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x,  1 < s \/ 1 = s (tor  46)
46)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
47)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
```

```
 1 = s,
 s < n ||-- A[s] = x,  s < n \/ s = n (aimpl  48,50)
48)  1 < k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,
 s < n ||-- A[s] = x,  s < n \/ s = n (tor  49)
49)  1 < k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,
 s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
50)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 = s,
 s < n ||-- A[s] = x,  s < n \/ s = n (tor  51)
51)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 = s,
 s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
52)  1 < k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x (TRIV)
53)  1 < k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 = s,
 s = n ||-- A[s] = x (aif  54)
54)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x (aimpl  55,64)
```

```
55)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x (anot  56)
56)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  (1 < s \/ 1 = s) /\
 (s < n \/ s = n) (tand  57,59)
57)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  1 < s \/ 1 = s (tor  58)
58)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
59)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  s < n \/ s = n (aimpl  60,62)
60)  1 < k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,
 s = n ||-- A[s] = x,  s < n \/ s = n (tor  61)
61)  1 < k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,
 s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
62)  1 < k,
 k < n,
```

```
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 = s,
 s = n ||-- A[s] = x,  s < n \/ s = n (tor  63)
63)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 = s,
 s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
64)  1 < k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x (TRIV)
65)  1 < k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k + 1 =>
  if A[i] = x then false else true fi) (aif  66)
66)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi)
 ||-- ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k + 1 =>
  if A[i] = x then false else true fi) (timpl  67)
67)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))
 ||-- forall(i:Nat) i < k + 1 =>
 if A[i] = x then false else true fi (anot  68)
68)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
```

```
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi)
 ||-- forall(i:Nat) i < k + 1 =>
 if A[i] = x then false else true fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tall  69)
69)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi)
 ||-- i < k + 1 => if A[i] = x then false else true fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (timpl  70)
70)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tand  71,92)
71)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s \/ 1 = s (tor  72)
72)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aimpl  73,90)
73)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (anot  74)
74)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
```

```
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,
 1 = s,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tand  75,88)
75)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s,  1 < s \/ 1 = s (tor  76)
76)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aimpl  77,82)
77)  1 < k,  k < n,  ~ (~ (((1 < s) \/ 1 = s) /\
 (s < n \/ s = n))), i < k + 1
 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (anot  78)
78)  1 < k,  k < n,  (1 < s \/ 1 = s) /\
 (s < n \/ s = n),  i < k + 1
 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aand  79)
79)  1 < k,  k < n,  1 < s \/ 1 = s,  s < n \/ s = n,
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aor  80,81)
80)  1 < k,  k < n,  1 < s,  s < n \/ s = n,
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (TRIV)
81)  1 < k,  k < n,  1 = s,  s < n \/ s = n,
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (TRIV)
82)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (tif  83,86)
83)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1 ||-- A[i] = x => false,  1 < s,  1 = s (timpl  84)
84)  1 < k,
 k < n,
```

```
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1,
 A[i] = x ||-- 1 < s,  1 = s (xrewrite  85)
85)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1,
 A[i] = x ||-- true,  1 = s (TRIV)
86)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1 ||-- ~ (A[i] = x) => true,
 1 < s,  1 = s (xrewrite  87)
87)  1 < k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1 ||-- true,  1 = s (TRIV)
88)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s,  s < n \/ s = n (xrewrite  89)
89)  1 < k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 true,  1 = s (TRIV)
90)  1 < k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (xrewrite  91)
91)  1 < k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
```

```
 i < k + 1 ||-- true,  1 = s (TRIV)
92)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 s < n \/ s = n (xrewrite  93)
93)  1 < k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||--
 if A[i] = x then false else true fi,  true (TRIV)
94)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k + 1 =>
     if A[i] = x then false else true fi
 fi (tif  95,148)
95)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- (1 < s \/ 1 = s) /\
 (s < n \/ s = n) => A[s] = x (timpl  96)
96)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n)
 ||-- A[s] = x (aand  97)
97)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
```

```
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 < s \/ 1 = s,
 s < n \/ s = n ||-- A[s] = x (aor  98,123)
98)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 < s,
 s < n \/ s = n ||-- A[s] = x (aor  99,111)
99)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 < s,
 s < n ||-- A[s] = x (aif  100)
100)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x (aimpl  101,110)
101)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x (anot  102)
102)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  (1 < s \/ 1 = s) /\
 (s < n \/ s = n) (tand  103,105)
103)  1 = k,
 k < n,
```

```
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  1 < s \/ 1 = s (tor  104)
104)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
105)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x,  s < n \/ s = n (aimpl  106,108)
106)  1 = k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,
 s < n ||-- A[s] = x,  s < n \/ s = n (tor  107)
107)  1 = k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,
 s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
108)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 < s,
 s < n ||-- A[s] = x,  s < n \/ s = n (tor  109)
109)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 < s,
 s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
110)  1 = k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s < n ||-- A[s] = x (TRIV)
```

```
111)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 < s,
 s = n ||-- A[s] = x (aif  112)
112)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x (aimpl  113,122)
113)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x (anot  114)
114)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  (1 < s \/ 1 = s) /\
 (s < n \/ s = n) (tand  115,117)
115)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  1 < s \/ 1 = s (tor  116)
116)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
117)  1 = k,
 k < n,
```

```
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x,  s < n \/ s = n (aimpl  118,120)
118)  1 = k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,
 s = n ||-- A[s] = x,  s < n \/ s = n (tor  119)
119)  1 = k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 < s,
 s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
120)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 < s,
 s = n ||-- A[s] = x,  s < n \/ s = n (tor  121)
121)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 < s,
 s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
122)  1 = k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 < s,
 s = n ||-- A[s] = x (TRIV)
123)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 = s,
 s < n \/ s = n ||-- A[s] = x (aor  124,136)
124)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
```

```
     if A[i] = x then false else true fi
 fi,
 1 = s,
 s < n ||-- A[s] = x (aif  125)
125)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x (aimpl  126,135)
126)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x (anot  127)
127)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x,  (1 < s \/ 1 = s) /\
 (s < n \/ s = n) (tand  128,130)
128)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x,  1 < s \/ 1 = s (tor  129)
129)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
130)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x,  s < n \/ s = n (aimpl  131,133)
```

```
131)  1 = k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,
 s < n ||-- A[s] = x,  s < n \/ s = n (tor  132)
132)  1 = k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,
 s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
133)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 = s,
 s < n ||-- A[s] = x,  s < n \/ s = n (tor  134)
134)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 = s,
 s < n ||-- A[s] = x,  s < n,  s = n (TRIV)
135)  1 = k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s < n ||-- A[s] = x (TRIV)
136)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi,
 1 = s,
 s = n ||-- A[s] = x (aif  137)
137)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x (aimpl  138,147)
138)  1 = k,
 k < n,
```

```
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x (anot  139)
139)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  (1 < s \/ 1 = s) /\
 (s < n \/ s = n) (tand  140,142)
140)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  1 < s \/ 1 = s (tor  141)
141)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  1 < s,  1 = s (TRIV)
142)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x,  s < n \/ s = n (aimpl  143,145)
143)  1 = k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,
 s = n ||-- A[s] = x,  s < n \/ s = n (tor  144)
144)  1 = k,
 k < n,
 ~ (~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))),
 1 = s,
 s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
145)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
```

```
 1 = s,
 s = n ||-- A[s] = x,  s < n \/ s = n (tor  146)
146)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 1 = s,
 s = n ||-- A[s] = x,  s < n,  s = n (TRIV)
147)  1 = k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 1 = s,
 s = n ||-- A[s] = x (TRIV)
148)  1 = k,
 k < n,
 if (1 < s \/ 1 = s) /\ (s < n \/ s = n)
   then A[s] = x
   else forall(i:Nat) i < k \/ i = k =>
     if A[i] = x then false else true fi
 fi ||-- ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k + 1 =>
  if A[i] = x then false else true fi) (aif  149)
149)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi)
 ||-- ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k + 1 =>
  if A[i] = x then false else true fi) (timpl  150)
150)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n))
 ||-- forall(i:Nat) i < k + 1 =>
 if A[i] = x then false else true fi (anot  151)
151)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi)
```

```
 ||-- forall(i:Nat) i < k + 1 =>
 if A[i] = x then false else true fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tall  152)
152)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi)
 ||-- i < k + 1 => if A[i] = x then false else true fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (timpl  153)
153)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tand  154,175)
154)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s \/ 1 = s (tor  155)
155)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aimpl  156,173)
156)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)),
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (anot  157)
157)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
```

```
 1 < s,
 1 = s,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) (tand  158,171)
158)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s,  1 < s \/ 1 = s (tor  159)
159)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aimpl  160,165)
160)  1 = k,  k < n,  ~ (~ (((1 < s) \/ 1 = s) /\
 (s < n \/ s = n))),  i < k + 1
 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (anot  161)
161)  1 = k,  k < n,  (1 < s \/ 1 = s) /\
 (s < n \/ s = n),  i < k + 1
 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aand  162)
162)  1 = k,  k < n,  1 < s \/ 1 = s,  s < n \/ s = n,
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (aor  163,164)
163)  1 = k,  k < n,  1 < s,  s < n \/ s = n,
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (TRIV)
164)  1 = k,  k < n,  1 = s,  s < n \/ s = n,
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (TRIV)
165)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (tif  166,169)
166)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1 ||-- A[i] = x => false,  1 < s,  1 = s (timpl  167)
167)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
```

```
 i < k + 1,
 A[i] = x ||-- 1 < s,  1 = s (xrewrite  168)
168)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1,
 A[i] = x ||-- true (TRIV)
169)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1 ||-- ~ (A[i] = x) => true,
 1 < s,  1 = s (xrewrite  170)
170)  1 = k,
 k < n,
 forall(i:Nat) i < k \/ i = k =>
 if A[i] = x then false else true fi,
 i < k + 1 ||-- true (TRIV)
171)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s,  s < n \/ s = n (xrewrite  172)
172)  1 = k,
 k < n,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 true (TRIV)
173)  1 = k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 1 < s,  1 = s (xrewrite  174)
174)  1 = k,
 k < n,
 A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- true (TRIV)
175)  1 = k,
```

```
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 s < n \/ s = n (xrewrite  176)
176)  1 = k,
 k < n,
 (1 < s \/ 1 = s) /\ (s < n \/ s = n) => A[s] = x,
 ~ (((1 < s) \/ 1 = s) /\ (s < n \/ s = n)) =>
 (forall(i:Nat) i < k \/ i = k =>
  if A[i] = x then false else true fi),
 i < k + 1 ||-- if A[i] = x then false else true fi,
 true (TRIV)
The proof consists of 176 nodes.


Q.E.D.
```

As mentioned, even though this proof runs relatively long at 176 steps, it is easily done in a minute or two if the user knows what to do; the user's work in this case is limited to applying a sequence of bpc and xrewrite. This proof is however not automatable by the simplest strategy for BPC proofs (with the command repeat [xrewrite, bpc]); at one point (branch 1.2 when performing the proof; step 94 above) bpc fails if xrewrite is applied first.

This proof is not very complicated; its length derives from from the size of the theorem. Consequently, performing this proof manually would presumably not be very difficult step for step, but would nevertheless be rather long-winded because of the sheer number of steps and the work necessary to be sure not to make trivial errors.